OS SUPPORT FOR HETEROGENEOUS MEMORY MANAGEMENT

A Thesis
Presented to
The Academic Faculty

by

Sudarsun Kannan

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology
August 2016

OS SUPPORT FOR HETEROGENEOUS MEMORY MANAGEMENT

Approved by:

Professor Ada Gavrilovska,
Committee Chair
School of Computer Science
Georgia Institute of Technology

Professor Karsten Schwan, Advisor
School of Computer Science
Georgia Institute of Technology

Professor Umakishore Ramachandran
School of Computer Science
Georgia Institute of Technology

Professor Moinuddin Qureshi
Electrical and Computer Engineering
Georgia Institute of Technology

Professor Remzi H. Arpaci-Dusseau
Department of Computer Sciences
University of Wisconsin-Madison

Dr. Greg Eisenhauer
School of Computer Science
Georgia Institute of Technology

Date Approved: 07/01/2016

To my Amma, Appa,

To my brother, my wife,

To my advisers Karsten, Ada,

To my family

ACKNOWLEDGEMENTS

First and foremost I would like to express my deepest gratitude and thanks to my advisers Prof. Karsten Schwan and Prof. Ada Gavrilovska without whom this dissertation would not have been possible. It has been an honor to be their Ph.D. student. Right from the day I joined the group as a Masters student, my adviser's willingness to motivate and guide me towards research and career directions has contributed greatly to this dissertation. I am indebted to them for providing the research freedom and patience without which most of the research would not have been possible. I learned the values of putting research first from Karsten and will greatly miss my interactions with him. His ajar office doors were a confidence booster. I am privileged to be advised by Ada, who has been a great strength of support until this day right from the day I started working on my master's project. Ada has been instrumental in shaping my every research idea including the numerous last minute paper commits. Karsten's sudden loss was a great shock to all of us. Ada filled the void and made this dissertation possible.

I would also like to thank Prof. Moinuddin Qureshi for advising me at various points in my research. Prof. Qureshi's 'Advanced Memory Systems' was one my favourite class at GT. I had the opportunity to work with him on a research problem and have learned several subtle but vital aspects of research and presentation of the research. I would also take this opportunity to thank Prof. Remzi H. Arpaci-Dusseau, who agreed to be on my dissertation committee and provided several valuable insights that played a significant role towards my last work for in this dissertation. I would like to thank Professor Sudhakar Yalamanchili, who was part of my several research discussions and provided valuable feedback. I would like to thank the other members of my dissertation committee, Prof. Umakishore Ramachandran, and Dr. Greg Eisenhauer. I would especially like to thank Greg for reviewing and providing valuable feedback on our papers. Special thanks to Dr. Mathew Wolf, and Dr. Jeff Young for the technical support at various points of this dissertation.

My mentors during my internships at Intel Labs, Sanjay Kumar, and my mentor at HP Labs, Dr. Dejan Milojicic, have had a significant impact on shaping the ideas presented in this thesis. Dejan thought me the importance of clarity in communication and presentation of ideas, and the value of fast progress in the systems research. Sanjay was very helpful in making me understand the practical benefits of a research approach, and providing the infrastructure, which is a critical aspect of systems research. I also gratefully acknowledge the funding sources, specifically, the Intel Labs ISRA grant that made my Ph.D. work possible including the opportunity to attend conferences. I would also like to thank Dr. Abhishek Gupta with whom I had several discussions during my internship at HP Labs.

I would like to express my heartfelt thanks to Professor Venkat, our Ph.D. adviser who was extremely receptive and provided a quick solution to various administrative and academic issues. He was extremely supportive during Karsten's demise. Special thanks to Susie McClain, our group admin, who has been extremely helpful in taking care of our travels and day to day lab needs.

My time at Georgia Tech was made enjoyable in large part due to many friends and groups that became a part of my life. I am grateful for the time spent and several intellectually stimulating discussions with my past and present labmates in the CERCS group over the years. A special mention to Minsung Jang, Hrishikesh Amur, Anshuman, Ketan Bhardwaj, Dipanjan Sengupta, Bharath Srinivasan, Mukil Kesavan, Vishal Gupta, Subramanya R. Dulloor, Min Lee, Fang Zheng, Hobin Yoon, Alex Merritt, and Jai Dayal. I am incredibly grateful to Dipanjan Sengupta and Krishnakumar Balasubramanian for being a great room-mates. I would also like to thank all my friends at various moments in my Ph.D. life.

Finally, I would like to thank my parents, wife, brother, and my family for their love, support, and encouragement over the years. A special thanks to my cousin, Shrinivas, who was just a call away. My elder brother Baranidaran has guided me through every step of my life without whom this dissertation would not have been possible. I thank my wife for providing the love and support in spite of her rigorous academic schedule. Finally, I would like to dedicate this thesis to my parents for what I am today is all because of them.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

The volume of data generated by servers and sensor-rich client devices are dramatically increasing, and this data volume poses a significant challenge in scaling existing memory and data storage technologies. At one end of the spectrum, scaling DRAM density without increasing the cost or energy is becoming more challenging. On the other hand, block-based storage with larger capacities, such as NAND flash, suffers from significantly lower bandwidth and high latency relative to an in-memory access. In response, to reduce the gap between computing, memory, and storage, system architects are exploring alternative and heterogeneous memory technologies such as byte addressable non-volatile memory, 3D-stacked memory, and others. These heterogeneous memory technologies vary significantly in bandwidth, latency, and capacity and therefore increase the operating system and application complexity, resulting in several resource management challenges such as inefficient cache use and energy overheads. This dissertation is motivated by the lack of operating system support to seamlessly scale memory capacity of applications across the heterogeneous memory and to also provide fast persistent storage that can efficiently use system resources such as processor cache and energy.

In the first part of this dissertation, we analyze the impact of heterogeneous memory on applications and discuss the limitations of state-of-the-art solutions in addressing the memory capacity scaling issues, persistent storage bottlenecks, and lack of operating system (OS) and hypervisor support for heterogeneous memory.

In the second part of this dissertation, we design persistent virtual memory (pVM), a system software abstraction that extends the OS virtual memory to provide applications with automatic OS-level memory capacity scaling, flexible memory placement policies, and fast persistent object storage using heterogeneous memory such as nonvolatile memory (NVM). Using pVM for memory scaling improves application performance by 2x compared to using state-of-the-art virtual filesystem-based (VFS) NVM solutions. pVM's virtual

memory-based design reduces TLB and cache misses by up to 80%, and up to a 4x reduction in the OS time for persistent object storage compared to several VFS-based NVM solutions.

We next extend this solution to design HeteroOS, a system software to support heterogeneous memory in virtualized datacenter systems. HeteroOS makes the guest-OSes heterogeneity-aware, supports application-transparent memory placement, and memory resource sharing across multi-tenant guest VMs. HeteroOS extracts guest-OS level information about how applications use virtual memory pages to provide automatic memory placement without page migrations. For further improving the performance benefits, HeteroOS provides a coordinated management between the hypervisor and the guest. Overall, HeteroOS improves the performance of applications by 194% compared to a naive approach of using only a slower memory, and up to 2x gains compared to the state-of-the-art approach that uses hypervisor exclusively for heterogeneous memory management.

In the third part of this dissertation, we discuss the system resource usage implications of heterogeneous memory. Specifically, byte addressable NVMs have high write latency and write energy. Although the processor cache plays a critical role in reducing direct writes to NVM, but dual use of NVM for capacity scaling and persistence introduces cache interference and contention issues. To address this, we first propose an OS-level and library-level allocator design that aims to reduce cache interference, and overall direct NVM writes resulting in improved application performance. For the OS-level, we propose a novel approach for contiguous memory page allocation that partitions cache for persistence and non-persistence use. As a result, the cache misses due sharing reduces by up to 10%. The NVM write-aware persistent allocator further reduces cache misses by 12% for end-user applications and around 4% for SPEC benchmark.

In the fourth part of this dissertation, we focus on several energy-aware persistence methods that provide data consistency and relaxed durability, without impacting application correctness. We first show that using strict consistency and durability persistence guarantees can increase the CPU and NVM energy usage by up to 7.1x and 4.2x, respectively. To address the high energy cost, we first propose a set of energy efficient durability

optimizations that trades performance for a reduction in the energy use. When such optimizations are insufficient in critically low energy state, we design and employ a relaxed durability approach that trades strict durability guarantees with up to a 3x CPU energy and 2x NVM energy usage reduction without affecting the correctness of an application.

In the final part of this dissertation, we discuss how application and service-level redesign for heterogeneous memory can significantly improve application performance and reliability. We explore applications from different computing environments – checkpoint-restarts for high-performance computing (HPC), web browsers for end-user devices, and finally, NoSQL databases used in data centers. We first discuss the design of an NVM-based checkpoint-restart that treats NVM as a virtual memory to reduce the application overhead of frequent checkpoints for HPC applications. Our approach reduces the checkpointing cost to a node local NVM by 15% and reduces the peak interconnect bandwidth usage by up to 45% for a remote checkpoint. We next examine the benefits of using NVM as a virtual memory to address the storage and computational needs of web browsers running in a sandboxed environment. Our technique of using the virtual memory page protection technique for NVM provides two benefits. First, browser applications can use NVM for a secure memory-based storage that avoids filesystem APIs in a sandboxed environment. Second, avoiding expensive filesystem APIs in a sandboxed environment improves storage performance by 2x. We finally discuss the impact of heterogeneous memory in the widely used NoSQL databases. We redesign an LSM-based NoSQL database to exploit memory heterogeneity and improve access parallelism. Our design improves read and write throughput by up to 3x compared to using a vanilla design that replaces an SSD with an NVM for persistence.

CHAPTER I

INTRODUCTION

The combined rapid increase in transistor density, platform-level parallelism, and use of large datasets in modern scientific and commercial workloads is exerting severe memory pressure on server systems. However, DRAM scalability, higher access latency, and lower bandwidth continue to be a bottleneck due to energy and cost limitations. This has led researchers to explore the use of alternative memory technologies which can supplement traditional DRAM by providing higher capacity (C), lower latency (L), and increased bandwidth (B). However, recent trends show that a single memory technology is not expected to solve all the bottlenecks (C, L, and B). For instance, byte-addressable nonvolatile memories (NVMs) such as phase change memory (PCM), are expected to provide 2x-4x higher memory capacity than DRAM, but can add around 5x higher write latency, and up to 5x-8x lower bandwidth. Similarly, on-chip stacked 3D-DRAM [52, 92, 102, 53] are expected to increase memory bandwidth by 8x-14x [52], but are expected to have limited capacity (2x-4x lower capacity than DRAM). As result, in order to provide applications with large capacity, low latency, and high bandwidth, future systems will support multiple levels of memory, (i.e.,) heterogeneous memory. Figure 1 shows a high-level view of heterogeneous memory in the system stack.

As a result of memory heterogeneity, it is obvious that the complexity of OS-level/software memory management will increase. Prior research has proposed hardware and software solutions that are limited by managing the unique needs of a single type of memory technology. Briefly, prior work such as [163, 92, 123, 102, 53] have proposed using on-chip 3D-DRAM as a last-level cache, whereas other research such as [79, 52, 64, 31] has discussed the advantages of using them as a H/W and OS-managed 'FastMem'. Similarly, prior research on NVMs can be classified into research that uses application-transparent methods for providing additional capacity benefits such as [125, 129], or hybrid solutions that exploit non-volatility

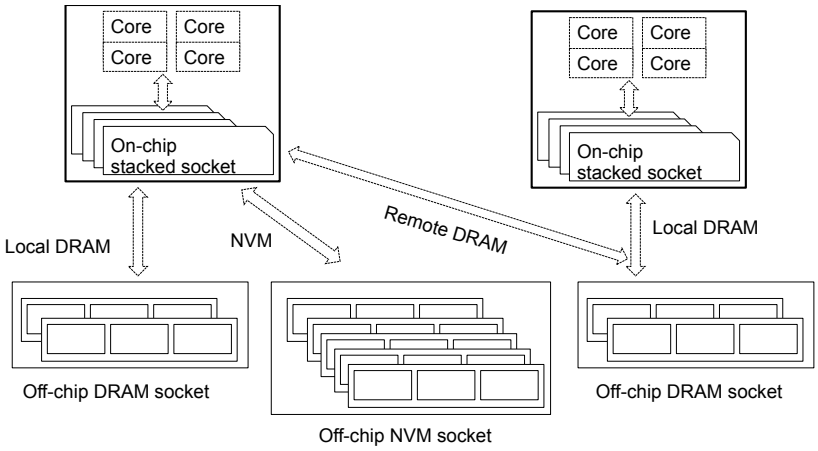


Figure 1: Heterogeneous memory hardware view

to provide fast persistence, and also additional capacity [146, 56, 129, 67].

Yet there has been little work to understand how future operating systems should manage these complex memory structures all at once: What should be the OS-level abstraction, and how should be the OS designed? Where to place which data, how or when to move it, and how to do so in the typically virtualized settings of cloud and datacenter workloads? We next discuss these questions in detail.

1.1 Statement of Problem

- **Automatic memory scaling across heterogeneous memory.** To compliment the increasing hardware support for memory capacity, the system software should support automatic memory scaling across these memories. Automatic scaling refers to the seamless use of one memory when resources of other memory are exhausted. Surprisingly, most state-of-the-art designs do not provide such capabilities. This dissertation aims to provide such seamless scaling.

- **OS-level abstraction.** To seamlessly scale memory, a unified OS-level abstraction is important. For example, current NVM-based solutions rely on the virtual file subsystem (VFS) to provide persistence and memory scaling [146, 56, 129, 67]. As a result, the virtual memory (VM) subsystem cannot manage or scale across NVMs. The abstraction is important for OS-level management also. This dissertation proposes a virtual memory-based NUMA node-like abstraction for heterogeneous memory, with

memory type-specific optimizations.

- **Cache and TLB efficiency.** Efficient use of the processor cache and TLB is a critical factor for application speedup, specifically, when there are capacity constraints for faster memory types. Cache and TLB efficiency not only depends on the application access pattern but also on how the OS-level data structures are designed and managed. For example, prior research has proposed extensions to VFS for enabling capacity-use from NVMs. However, the VFS data structures originally designed to support block-based disk, are highly cache and TLB inefficient, imposing a severe performance penalty on applications. This dissertation shows that by embracing the virtues of a VM-based design for different memory devices, the cache and TLB efficiency can be improved significantly.

- **Flexible memory placement policies for applications.** The OS should provide transparent scaling, but more importantly, users/applications should have the flexibility for data placement across different memory types. Although current solutions provide direct allocations to a specific memory type, they lack the capability to support policies (like for NUMA platforms) that enable application developers to effective use the heterogeneous memory.

- **Support for memory heterogeneity in virtualized systems.** It is a well-known fact that most large-scale applications currently run in virtualized datacenters. Few proposals [79] exist for memory heterogeneity management in virtualized systems. In such proposal, the heterogeneity is transparent to guest-OSes and applications, and all the management is done at the hypervisor-level. However, the hypervisor has limited information about application resulting in an inefficient use of memory heterogeneity.

- **OS-level virtual memory information for prioritizing memory placement.** In a multi-tenant environment, relying on applications for managing or providing data placement hints is not practical. Moreover, they lack the holistic view of the system resources. Hence, it is critical for the OS and the hypervisor to manage memory placement across heterogeneous memories, given their significant capacity, latency,

and bandwidth differences. Current solutions just rely on expensive migrations across memory types neglecting the rich OS-level information about applications' memory use. This dissertation illustrates a design for extracting and managing OS-level memory placement with a goal to reduce migrations.

- **Improved persistent storage.** Heterogeneous memory such as NVM provides not only more capacity but also fast persistence. While NVMs are expected to improve storage performance, when using traditional filesystems the cost of the software overheads continues to dominate. Specifically, for applications that required simple object storage without the need for a filesystem hierarchy, current approaches add significant software overheads. This dissertation explores methods to reduce such software (OS-level) cost for a wide range of applications, without compromising the consistency or durability guarantees.

- **Cache and energy efficient persistence.** Future memories such as NVMs can support additional capacity and persistence, referred to as "dual use" in this dissertation. However, supporting persistence requires both the OS and high-level libraries to provide atomic, consistent, isolated, and durable (ACID) guarantees to the application. Providing ACID guarantees requires constantly flushing data from the processor cache, and also requires logging data similarly to traditional filesystems. Consequently, this results in cache interference for dual use and higher NVM writes. NVM technologies have high active energy consumption, and an increase in NVM writes results in significant energy overheads. This dissertation develops methods for improving the efficiency of processor cache and for lowering of the energy consumption for persistence use.

- **Application-level redesign for heterogeneous memory.** Finally, this thesis also explores application- and service-level redesign for efficient use of heterogeneous memory. We study applications that include high-performance computing checkpoint/restart, web-browsers for end-user devices, and NoSQL databases widely used in datacenter applications.

1.2 Thesis statement

To maximize the application performance benefits and achieve efficiency on future platforms with heterogeneous memory devices, it is important to introduce OS-level abstractions and mechanisms for heterogeneous memory management solution that enable seamless memory capacity scaling and memory placement across heterogeneous memory, provide fast persistent object storage, and lead to the efficient use of system resources. For additional performance and reliability gains, application and service-level redesign for heterogeneous memory is critical.

1.3 Contribution

We describe the main contributions of this dissertation:

- We design persistent virtual memory (pVM), a design that treats heterogeneous memory such as NVM as a memory node and extends the OS VM subsystem for memory capacity scaling, and fast and persistent object stores. pVM's VM-based design provides automatic memory capacity scaling, support for flexible NVM data placement, fast page access, and improved processor cache and TLB efficiency. pVM also extends the VM data structures to provide fast, consistent, and durable object storage. pVM's memory scaling improves application performance by 2x compared to using state-of-the-art virtual filesystem-based (VFS) NVM solutions. pVM's virtual memory-based design reduces TLB and cache misses by up to 80%, and the persistent object storage design reduces the OS time by up to 4x compared to state-of-the-art VFS-based solutions.

- We design HeteroOS, an OS design that extends pVM for providing memory heterogeneity support in virtualized environment. HeteroOS makes guest OSes heterogeneity-aware by extending the existing NUMA support for OS to guest VMs. Further, HeteroOS exploits the guest-OS awareness to extract information about applications' memory page use to provide application-transparent smart memory placement across

different memory types depending on their bandwidth and latency. HeteroOS also extends the hypervisors to support efficient resource management across multiple VMs. Overall HeteroOS improves the performance of applications by 194% compared to a naive approach of using only the slower memory, and up to 2x compared to state-of-the-art hypervisor exclusive approaches.

- Applications can use NVMs for both capacity and persistence which we term as dual use. We analyze the detrimental impact of cache sharing for dual use of NVM. We propose a novel OS-level cache partitioning to reduce cache interference thereby reducing NVM writes. We also explore persistent memory library allocator-level optimizations. For the OS-level, we propose a novel approach for contiguos memory page allocation that partitions cache for persistence and non-persistence use. As a result, the cache misses due sharing reduces by up to 10%. The NVM write-aware persistent allocator further reduces cache misses by 12% for end-user applications and around 4% for SPEC benchmark.

- Supporting ACID-based persistence for NVMs incurs significant energy cost. We propose energy-aware persistence (EAP), a design that provides energy efficient durability, and a relaxed durability for critically low energy states. We show that using strict consistency and durability can increase CPU and NVM energy usage by up to 7.1x and 4.2x, respectively. To address the high energy cost, we propose an energy efficient durability and relaxed durability that reduce the CPU and NVM energy usage by up to 3x and 2x, respectively.

- Finally, we explore the different application and service-level redesign to support and exploit benefits of memory heterogeneity. We demonstrate the application performance improvement by redesigning checkpoint-restart service used in HPC applications, Web browsers used in almost all end-user devices and NoSQL databases used extensively in cloud-based applications. Our checkpoint approach reduces the checkpointing cost to a node local NVM by 15% and reduces the peak interconnect bandwidth usage by up to 45% for a remote checkpoint. Our approach of using NVM

as a virtual memory for browser-based storage improves storage performance by 2x. Finally, our redesign of an LSM-based NoSQL database for exploiting memory heterogeneity and improve parallelism provides up to 3x higher throughput gains compared to a vanilla design that just replaces an SSD with an NVM for persistence.

1.4 Organization

The remainder of this dissertation is organized as follows. In Chapter 2, we provide a background of different heterogeneous memory technologies and their usage models explored by prior research. We next discuss the heterogeneous memory emulation method followed by a detailed analysis of state-of-the-art solutions and performance characterization of large-scale applications.

In Chapter 3, we detail the design of persistent virtual memory (pVM), a system that provides seamless memory capacity scaling across heterogeneous memory. The chapter also describes the design for extending virtual memory for supporting object-based persistence for heterogeneous memory such as NVM.

In Chapter 4, we describe the OS and hypervisor design to support heterogeneous memory in virtualized datacenters. We also discuss our design for making the guest-OS heterogeneity-aware, methods to provide automatic OS-level memory placement, and finally, enable heterogeneous memory sharing across multiple VMs.

In Chapter 5, we analyze the cache implication of the dual use of NVM for additional capacity and persistence and propose solutions to reduce cache interference. In Chapter 6, we also address the energy overheads of memory-based persistence.

In Chapter 7, we explore the application and service-level redesign for supporting heterogeneous memory use. In Chapter 8, we discuss the related work, and finally in Chapter 9, we conclude this dissertation and present ideas for future research.

CHAPTER II

MOTIVATION

In this chapter, we first provide a background of different heterogeneous memory technologies. We then analyze the latency, bandwidth, and capacity impact of heterogeneous memory, and finally, discuss the drawbacks of state-of-the-art solutions.

2.1 Background of heterogeneous memory technologies

Ongoing trends with data-centric applications pose increased demand on the memory systems. At the same time, scaling issues with DRAM are forcing architects to redesign systems with heterogeneous memory devices. On-chip memory technologies such as High Bandwidth Memory (HBM) [52, 107] are expected to provide high bandwidth but can have limited capacity. In contrast, devices such as phase change memory (PCM) and STTRAM are expected to have higher density but suffer from lower bandwidth, higher latency, and lower lifetimes. We next discuss the software usage models of these memory technologies.

2.1.1 Byte addressable NVM usage models

Byte-addressable NVMs such as PCM [98, 125], memristor, and 3D XPoint [7], are expected to provide 100x faster read and write performance compared to current SSDs [57, 67, 47]. Table 16 shows their hardware characteristics compared against DRAM and NAND devices. These NVMs are expected to scale 2-4x more than DRAM because they can store multiple bits per cell without using refresh power and have known limitations imposed by an endurance of a few million writes per cell. Further, they can be connected to the memory bus, and accessed using load and store operations, which use the processor cache to reduce access latency [105]. Researchers have explored different usage models for the capacity and persistence use of NVM. Early NVM system software and hardware research [125, 98] studied the feasibility of using NVMs as a data cache for additional memory capacity that is

Table 1: Heterogeneous memory characteristics [52, 126]

Property	Stacked 3D (ns)	DRAM (ns)	PCM (ns)
Density	1x	4x-16x	16x-64x
Load, store lat (ns)	30-50	60	150,300-600
Bandwidth (GB/sec)	120-200	15-25	2

transparent to the application. These studies do not use NVMs for persistent storage. Recent software solutions such as [47, 57, 67] have redesigned the block-based filesystem to suit the memory-based storage by extending the VFS data structures. In contrast, research proposals such as Mnemosyne [146], NV-Heaps [56], and Aerie [145] have extended the ideas of the well-known LRVM work [136] to support a heap-based persistence. Interestingly, all current heap-based proposals are managed by the VFS. In contrast, pVM manages the NVM by extending the VM subsystem instead of the VFS. Our solution lets the system and applications use NVM both for capacity and for a heap-based object storage [146, 56, 145]. For comparison, we use Intel's PMFS [67] as the state-of-the-art solution due to several OS-level optimizations and its wide acceptance in the NVM research community. We also evaluate other approaches such as Mnemosyne and ramFS.

2.1.2 High bandwidth 3D DRAMs

Memory technologies such as Hybrid Memory Cube (HMC), High Bandwidth Memory (HBM) [52, 107], and Wide I/O(WIO), provide high bandwidth and DRAM-like memory array access latency shown in Table 1, but are projected to have limited capacity (1GB-8GB) compared to commodity DRAM (i.e., DDR3, and DDR4) [52]. This makes it difficult to completely replace commodity DRAM. Because of limited scaling, several prior research propose using them as a last-level (L4) cache [53, 91]. However, a 4GB can high tag space overhead and requires re-engineering of the cache design for tag placement plus accompanying changes in memory controller logic regarding timing, scheduling commands and address bus [107]. In contrast, recent proposals argue for their use as fast OS-managed FastMem [79, 107]. Considering that each of these technologies have some advantages and disadvantages in terms of capacity, latency, and bandwidth, the indications are clear that

Table 2: L:x,B:y indicates the latency increase factor x, and bandwidth reduction factor y respectively.

Factor	L:1, B:1	L:2, B:2	L:5, B:5	L:5, B:9
Latency (ns)	98	203	616	960
Bandwidth (GB/s)	22	11	3.7	1.2

for achieving maximum benefit support for memory heterogeneity is critical, which increases the complexity of OS/S/W management.

2.1.3 Heterogeneous memory emulation

The lack of commercially available heterogeneous memory components, other than DRAM, presents a methodological challenge for a meaningful study to understand the impact of heterogeneity in datacenter applications. Using cycle-accurate instruction-level simulators [52, 107, 64] for end-to-end (user and the OS stack) analysis of long-running applications is not practical. Therefore, we consider two generic types of memory.

- FastMem: high bandwidth and low latency, and limited capacity memory
- SlowMem: high latency and low bandwidth, but large capacity memory

To emulate FastMem and SlowMem, we use thermal throttling to dynamically vary the bandwidth of a DRAM socket by modifying PCI thermal register values. The bandwidth and latency impact of throttling on applications and the OS is based on their memory access pattern and does not have other side effects as studied by prior heterogeneous memory research [82, 79, 94]. For our study, we use the DRAM bandwidth and latency as the FastMem baseline, and reduce the bandwidth of SlowMem by up to 12x, and latency by up to 5x. However, the principle contributions of this work concerning the OS design, virtualization and management of heterogeneous memory are generic, and apply for other memory-technology parameters.

2.2 Application sensitivity towards memory heterogeneity

An OS-based memory heterogeneity management has to be effective for different class of applications that are memory, compute, storage, and network intensive. Hence, we first

study the impact of memory heterogeneity in different class of applications shown in Table 14 which is in contrast to prior work that have focused mostly on memory intensive applications. The applications analyzed include graph analytics, in-memory data stores, map-reduce-based computations, as well as traditional databases and web servers [69] as well as a persistent object storage application Snappy compression which we will discuss shortly.

Although we primarily use these server-based applications for analysis in this chapter, this dissertation also explores the implications of heterogeneous memory in end-user and HPC platforms. Hence, in Chapter 5, 6, 7, we introduce and discuss several end-user and HPC applications. Figure 4 shows the memory page distribution and the total memory pages used for the same set of applications. We next discuss the impact of memory bandwidth, latency, and capacity of heterogeneous memory on applications. These analysis motivates the need for OS-level, hypervisor, and application-level support for heterogeneous memory.

2.2.1 Latency and bandwidth impact

Figure 2 shows the performance reduction of different applications by increasing memory latency and reducing bandwidth in x-axis. The table in Figure 2 shows the memory intensity of applications in terms of misses per million instructions (MPKI).

Observations. As expected, application with higher MPKI have high impact from bandwidth and latency reduction. Interestingly, for most applications except graph analytics (X-Stream and Graphchi), the impact of memory bandwidth reduction is relatively less compared to the increase in memory latency. For instance, when the latency is reduced by 5x, the throughput of Redis and Memcached reduces by up to 2x, and 1.5x respectively. However, with further reduction of only the bandwidth, but same latency, the impact is relatively small. Similar results can be observed for in-memory mapreduce application Metis. For X-Stream and Graphchi, with high random access shown by the MPKI values, both latency and memory bandwidth impact performance.

For I/O intensive LevelDB, the throughput reduces only by 30% when the latency is

Table 3: Applications for heterogeneous memory memory analysis.

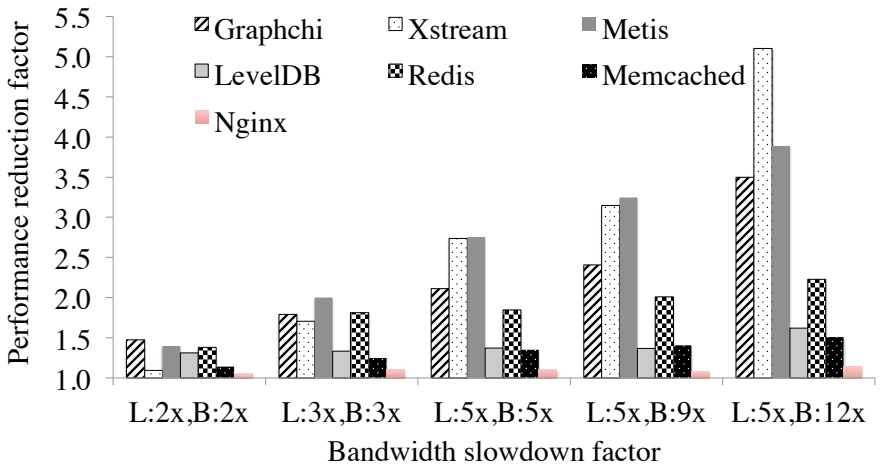
Application	Type	Description and Workload	Evaluation Metric	SlowMem usage type
GraphChi [96]	Graph analytics	Uses Orkut social graph input, 3 million nodes, 117 million edges, requires 8 GB memory [69]	Time (seconds)	Capacity
X-Stream [135]	Graph analytics	Edge-centric graph processing and uses same input as GraphChi	Time (seconds)	Capacity
Metis [44]	Data analytics	Shared memory mapreduce that optimizes Phoenix [130], 2GB crime dataset, 8 mapper-reducer threads	Time (seconds)	Capacity
LevelDB [4]	NoSQL database	Google's DB for bigtable, SQlite bench with 1M keys	MB/s	Capacity
Redis [20]	Data caching	Popular in-memory key-value store with support for persistence, Redis benchmark with 2M operations, 80% get	Requests/sec	Capacity
Memcached [14]	Data caching	In-memory key-value store, 200 sec memslap benchmark run, 80% get	MB/s	Capacity
NGinx [17]	Web serving	Popular webserver, 1 million static,dynamic, images webpages	Requests/sec	Capacity
FaceRec [19]	Image recognition	OpenCV face recognition on Gallagher dataset [71] 1.2GB input DB (2K images, each 800KB)	Time (seconds)	Capacity
Dedup [41]	Deduplication	Parsec Dedup benchmark 4GB OS image file	Time (seconds)	Capacity
Snappy [23]	Compression	Fast data compression used in several Google products, image, video, audio, document files -2GB	MB/s	Object store

reduced by 3x, but with 5x, interestingly, the throughput reduces by 68% and remains stable even with bandwidth reduction of 9x. This is because, for I/O intensive applications that use buffer cache in short burst, suffer from memory access latency. The memory efficient Nginx Web-server with maximum memory usage of 400MB-500MB has minimal impact due to latency or bandwidth. Although the Web servers have high I/O time, the buffer cache usage for serving a web page of few kilobytes is significantly less.

Insights. we learn two important insights. (1) Applications exhibit high variability in their sensitivity towards memory speeds, demonstrating the importance of software/OS controlled memory management that adapts to the workload characteristics. (2) The memory latency, and bandwidth not only impacts applications that are memory intensive, but also storage and network intensive. Note that, we also verified all the observations in the Intel

Table 4: Application working set size (WSS) and MPKI.

Applications	WSS (GB)	Allocated (GB)	In use (GB)	MPKI
Graphchi	2.8	21	5.8	27.4
X-Stream	2.9	2.9	2.9	24.8
Metis	5.1	7.6	7.1	14.9
LevelDB	0.14	0.6	.32	4.7
Redis	1.8	4	2.4	11.1
Nginx	0.06	0.11	0.08	2.1



	Graphchi	Xstream	Metis	LevelDB	Redis
MPKI	27.4	24.8	9.61	4.7	20.1

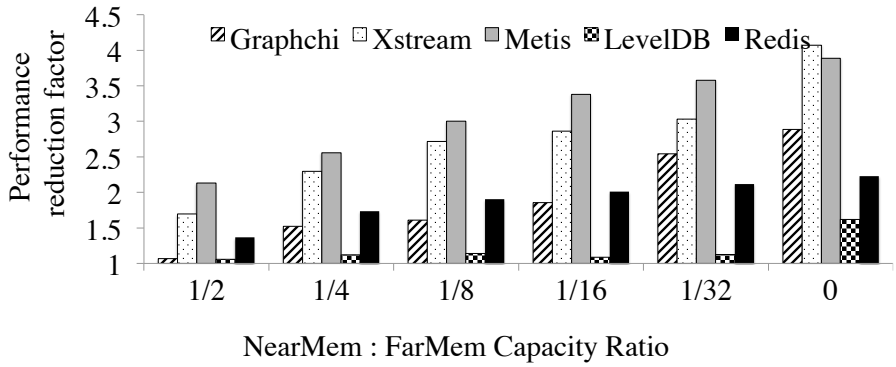
Figure 2: Latency and bandwidth sensitivity. In x-axis L:x,B:y represents latency increase factor x, and bandwidth reduction factor y relative to FastMem(L:1x,B:1x). Y-axis - performance reduction relative to FastMem.

NVMEP emulator where bandwidth and latency can be modified independently, and found similar trends.

2.2.2 Heterogeneous memory capacity impact

Figure 3 analyzes the impact of FastMem capacity. The y-axis shows the ratio of FastMem to to SlowMem capacity. Table in the Figure 3 shows the FastMem allocation miss-ratio [150] for the 1/2, 1/4, and 1/8 capacity ratio. The miss ratio is the total FastMem page allocation misses to the actual page allocations for a workload.

Observations. For 1/2 capacity ratio most applications except X-Stream, and Graphchi



Capacity vs. Miss Ratio	Graphchi	Xstream	Metis	LevelDB	Redis
1/2	0.3	0.3	0.6	0.0	0.2
1/4	0.7	0.8	0.7	0.0	0.5
1/8	0.8	0.9	0.8	0.0	0.7

Figure 3: Impact of FastMem capacity.

experience less than 30% overhead. This indicates that (1) either the actual number of allocated and used pages is less than FastMem capacity, or (2) applications allocate and release memory frequently.Reducing the capacity further to 1/4 increases the application slowdown not only for memory intensive applications like Metis, but also I/O buffer cache intensive applications like X-Stream, GraphChi [96]. This is because current OS prioritize heap allocations with most I/O pages allocated from slow memory. LevelDB with short transactions and use of buffer cache only for read operations (writes are flushed,synced) shows less that 15% up to 1:16 ratio.

Insights. Large scale applications frequently allocate/release memory. Providing flexible memory placement mechanisms that not only focus on heap but also buffer cache allocations can increase application performance.

2.3 State-of-the-art capacity scaling limitations

We next discuss the lack of (1) OS-level seamless capacity scaling, (2) application-transparent memory placement across heterogeneous memory, and (3) lack of heterogeneity awareness in virtualized systems.

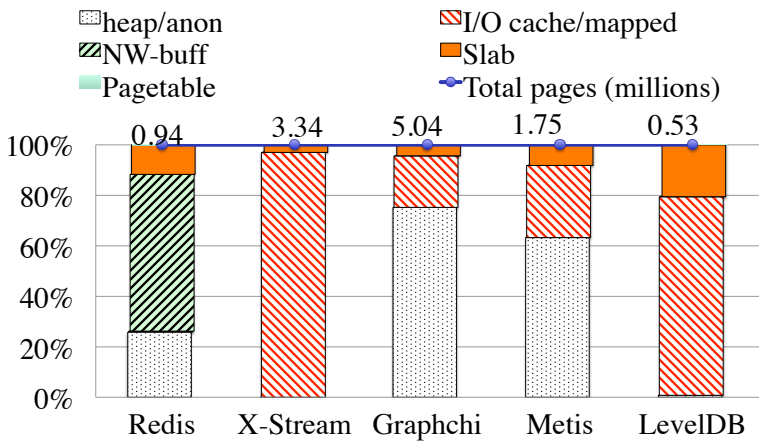


Figure 4: Application memory page distribution.

2.3.1 Lack of capacity scaling, and TLB, cache inefficiency

Several recent hardware and software research have discussed support for heterogeneous memory [98, 56, 67, 107, 125, 53, 52] in a single computer system. Proposals such as [98, 52, 107, 125] rely on page migration techniques across fast and slower memory. Proposals such as [56, 67] have proposed memory device specific management methods. Surprisingly, most solutions either do not seamlessly scale across different memory devices or do not provide any smart memory allocation (placement) mechanism, and only rely on expensive page migration techniques. We first discuss the device specific solutions followed by the drawbacks of migration-based mechanisms.

Recent research such as PMFS [67] propose a VFS-based management of the SlowMem (NVMs). Note that these solutions not only use SlowMem for storage, but to also increase the memory capacity of the applications (e.g., via mmap with MAP_PRIVATE to the VFS). However, a significant limitation of using a VFS for additional memory capacity is that it cannot seamlessly scale across different memory devices such as FastMem, or DRAM. This is because a VFS can only manage a filesystem, and whereas the virtual memory cannot manage a region managed by the VFS. This incompatibility results in lack of memory scaling across heterogeneous memory resource. Hence, applications that exhaust one memory type (e.g., FastMem) cannot seamlessly switch SlowMem or vice versa.

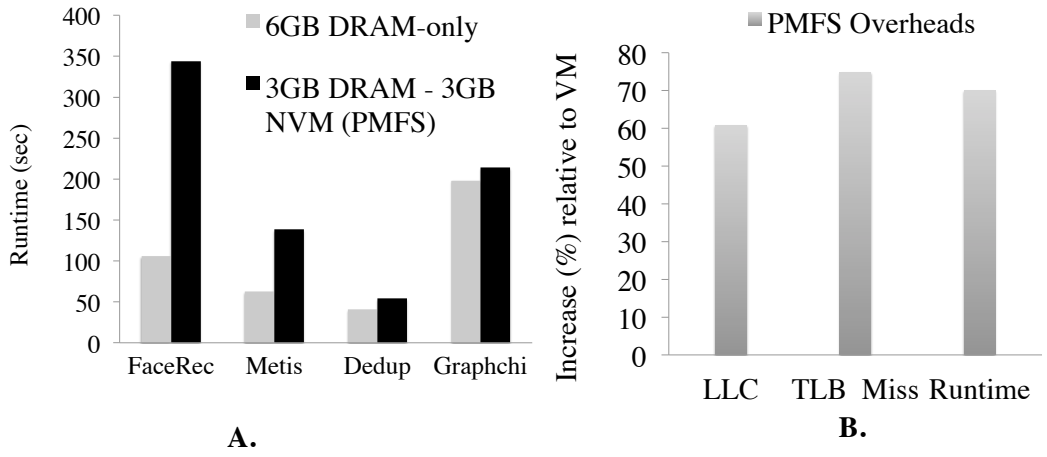


Figure 5: Capacity use analysis. (A) Performance under limited DRAM capacity, (B) Metis software overheads when using PMFS exclusively for capacity.

2.3.2 Lack of heterogeneous memory support in virtualized systems

The current virtualized system software is designed for homogeneous memory. The hypervisors neither expose memory heterogeneity nor allow flexible use of heterogeneous memory by guest-OS, and with complete control of memory management. Hypervisors view an entire guest-OS as an application, with minimal or no information about how applications use memory. Lack of information about applications use of memory limits the hypervisor's to use reactive management techniques - monitor hotness of pages, and migrate them accordingly. Lack of coordinated management between the guest-OS and VMM restrict several opportunities such as on-demand allocation to the right memory without the need for migration, or using OS semantics for memory placement.

2.3.3 Expensive hotness-based migrations

Several recent proposals such as [98, 52, 107, 125] have proposed hotness-based page migration techniques across heterogeneous memory types. While this provides seamless scaling, hotness-based page migrations can be significantly expensive. A hotness tracking mechanism has to periodically scan an application page table, disable the reference bit of all the pages in the page table, and for each page accessed subsequently, the access counter of the pages has to be incremented. When a counter reaches a threshold, the pages have to be

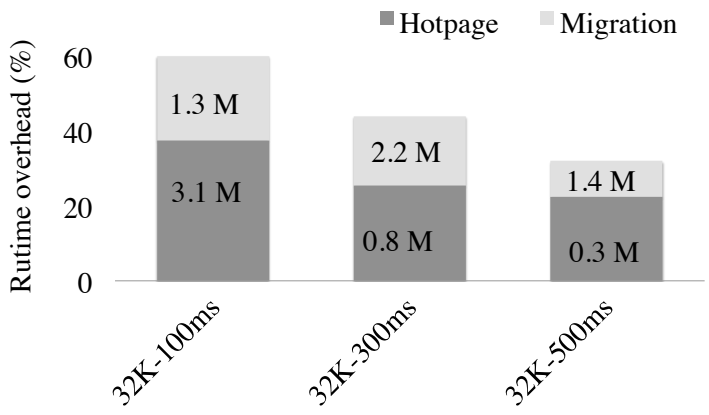


Figure 6: Graphchi hotness tracking and migration cost. Values in bars indicate pages tracked and migrated (in millions).

migrated. First, scanning the entire page table for an application periodically can induce high overhead. Note that failure to scan and migrate in a timely manner would mitigate all the benefits. Next, migrations are significantly expensive not just because of the data movement, but also from the cost of page table update, TLB invalidation and cache pollution. Figure 6 shows, even using a 100ms interval for 32K pages hot page detection shows up to 30% overhead, which only decreased by 10% for a 500ms scan interval. These drawbacks show that hotness-based migration techniques are insufficient.

2.3.4 Persistent storage limitations

NVM filesystem software overheads. Prior research has extensively studied the overheads of the block-based filesystem when using NVMs [145, 67, 56, 146]. The sources of overheads include the frequent kernel mediation required for handling I/O system calls, metadata updates (superblocks, inode bitmaps, inodes, and other data structures are updated before a block is modified), metadata concurrency issues from locking, and finally namespace management. Recent industry proposals have redesigned the OS filesystem metadata to track pages instead of blocks, remove the buffer caches, and provide cache-line size atomic updates [67]. Other research, such as Aerie [145] has proposed a decentralized approach by separating the trusted filesystem operations (e.g., permission check, metadata

Filesystem time: 60.5%	
Metadata Lock	5.16%
Metadata update	12.46%
copy_from_user	9.58%
copy_user_nocache	9.63%
Page fault	6.11%
Allocation	25.63%

Figure 7: Time spent (%) in the filesystem for Snappy application.

update, and integrity) from the untrusted operations (e.g., dentry cache and inode namespace management) and moving the untrusted operations to the user-level. This reduces the kernel mediation cost and cache pollution.

However, these optimizations do not eliminate the software overheads associated with frequent operations on small files. To analyze an I/O intensive application, we use a file compression service – Snappy [23] as a case study in Figure 7. In this example, Snappy compresses around 2GB of image, video, email, and document files. The input shows high variation in file sizes (kilobytes to gigabytes) and requires frequent user-to-kernel transitions in the form of system calls. The transitions are required mainly because a new output file is created for each input file, which stresses the filesystem metadata updates. Figure 7 shows that close to 60% of the time is spent in the OS filesystem. The table also shows a breakdown of the cost of the filesystem components, with significant portions attributed to metadata updates and locks, kernel-to-user buffer data copying, and kernel mediation. Even using the *mmap()* and *munmap()* interface does not solve the problem for small files as every *mmap()* call is supported by other system calls (*open()*, *close()*, *stat()*) and can aggravate the issue. *These results show that the OS filesystem overhead can be significant even in the state-of-the-art NVM filesystem.*

Object storage challenges. Existing Linux object-based filesystems such as Ceph [49] and S3fuse [21] offer applications an object-based put/get interface at the user-level but

```
Image **imgdb = nvroot_alloc("img_root", size);

START_TRANS (imgdb )
  for each new image:
    Image *imgdb[cnt]= nvmalloc("imgname", size, imgdb)
    cnt++
   ……
    /*Commit by flushing and logging */
    nvcommit("img_root", size, log=true)
END_TRANS (imgdb )

   /* persistent read, implicit load of all child ptrs*/
   img = nvread ("img_root", &size);
   /* non persistent NVM memory allocation */
   tmp = npmalloc(size)
```

Figure 8: Persistent object programming.

map objects to files internally at the filesystem level. Hence, frequent operations on small files or objects suffer from filesystem overheads. Prior work [145, 56, 146, 87] has proposed nonvolatile heap objects that reside in some larger mapped (using *mmap()*) persistent region, and can be accessed and modified with a load and store interface. Figure 8 shows the programming model for allocating, modifying, and committing a simple persistent image object using a persistent heap-based NVM library [87]. Although the persistent storage interface avoids direct filesystem interaction and mediation, the mapped regions encapsulating persistent objects are still managed by the VFS. As discussed earlier, this results in high page access latency as well as cache and TLB inefficiency. Another minor drawback of existing persistent object stores is that all persistent objects of an application are mapped to a single large region [87]. As a result, in order to retrieve even one object, the object store has to search the entire mapped region, thus increasing the average retrieval time.

2.3.5 Persistent write cache and energy issues

NVM's limitations of slow writes and high write energy are magnified for applications that require atomic, consistent, isolated and durable (ACID) updates. For maintaining ACID guarantees for persistence, the application not only needs to do extra writes to NVM but also

needs to perform a significant number of extra instructions for performing NVM writes in a transactional manner. As a result, persistence results in higher expensive NVM writes, cache interference when using NVM for both capacity and persistence, and more importantly, the overall system energy consumption also increases. As we discuss in chapter 5 and chapter 6, our analysis shows that a system that maintains persistence incurs a CPU energy increase of 7.3x and NVM energy increases of 5.1x compared to a baseline that does not support ACID guarantees. For computing platforms (such as mobile devices) where energy consumption is a critical factor, it is important that the energy cost of persistence is reduced.

CHAPTER III

OS SUPPORT FOR HETEROGENEOUS MEMORY

In the previous chapter, we discussed the lack of OS support for capacity scaling across heterogeneous memory such as DRAM and NVM, and for efficient support for NVM use for persistence. In this chapter we address the capacity scaling limitations and lack of fast persistent object storage of state-of-the-art solutions. We propose **persistent Virtual Memory (pVM)**, an alternative approach that extends the virtual memory (VM) subsystem, instead of the VFS, for achieving the dual benefits of memory capacity scaling and persistent storage using NVMs.

3.1 Heterogeneous memory OS support with pVM

Persistent virtual memory (pVM) is a design based on the principle that byte-addressable NVMs resemble 'slower' memory placed in parallel to DRAM with the memory bus as a hardware interface, rather than a faster disk. Hence, extending the virtual memory (VM) subsystem is better suited to more efficiently exploit NVMs for both capacity and persistent object storage. pVM is not a replacement for traditional filesystems and has been specifically designed for the use of NVM for scaling memory and providing a persistent object store. To the best of our knowledge, pVM is the first OS-based design that extends the VM subsystem to provide such dual-use.

pVM's OS hardware abstraction treats NVM as a NUMA node to which applications can transparently or explicitly allocate additional heap memory, as well as store persistent objects by using user space memory persistence libraries [146, 141]. This generic NUMA node-based design can support several NVM nodes, permit seamless scaling of memory capacity across the nodes, and provide applications with flexible NVM-specific memory placement policies that can be easily integrated with existing memory placement libraries. Unlike the VFS-based approach that maps both capacity and persistence allocations to a file, pVM's OS virtual memory framework distinguishes between persistent and non-persistent

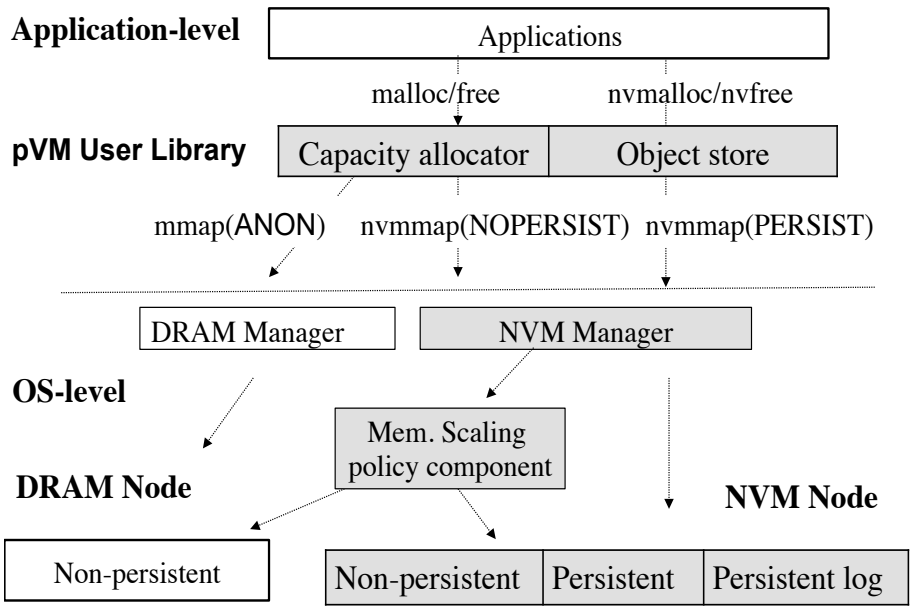


Figure 9: pVM – High level design. Shaded blocks indicate pVMs software-level stack changes.

memory (page) allocations and manages them independently without adding any persistent metadata management cost for capacity (heap) allocations. This significantly reduces page access time and cache and TLB misses, which is critical for applications' performance and helps mitigate NVMs' higher device access cost when compared to DRAM (see Table 16).

pVM also extends the VM subsystem with persistence management. This avoids the metadata complexity of a block-based filesystem and the overheads of kernel mediation from using a POSIX interface, which leads to reduced time spent in the OS. With pVM, persistent objects are mapped to a region of NVM pages that can be retrieved across application sessions. Performance of the persistent object store is improved with pVM due to fast on-demand page allocations and efficient TLB and cache use. pVM is designed for object stores with flat namespaces, such as NoSQL databases, personal user data, search engines, key-value stores, etc., and thus may not be a good fit for applications with significant dependence on the hierarchical filesystem. Hence, we envision that pVM's object store will coexist with a standard block-based NVM filesystem.

3.2 pVM overview

Setting. pVM is designed for byte-addressable NVM technologies such as PCM [125, 18, 109] and 3D Xpoint [7] that are connected with a memory controller and are placed at the same level in the memory hierarchy as DRAM. NVMs can be accessed by the CPU/applications with load-store instructions to NVM pages through a hardware supported page table [30], and they exhibit higher access latency and provide lower bandwidth compared to DRAM. Because byte-addressable NVMs are not yet commercially available, we emulate them using a dedicated NUMA socket and apply memory thermal throttling to reduce the bandwidth by up to 10x relative to DRAM. To account for the differing read versus write latencies, we inject software delays using a model that accounts for LLC load and store misses. We use the model proposed by [67, 139, 57]. These emulation methods are not required for commercially available NVDIMM technology (DRAM in the front backed by an SSD in the back), but NVDIMMS have a PCIe interface rather than a memory controller-based interface, and more importantly, NVDIMMS are limited by the DRAM scalability limits.

Approach overview. We design and develop pVM to provide applications with the following important capabilities: (1) *memory capacity scaling* – OS support to achieve application-transparent memory capacity scaling similar to a multi-socket NUMA machine, (2) *efficient persistence* – fast heap-based persistent object store for applications not dependent on the filesystem hierarchy, and (3) *improved performance* – reduced cache and TLB misses when using NVMs, and maximized application performance. Figure 62 illustrates a high-level overview of the pVM design.

For scaling capacity, pVM treats NVM as a separate NUMA node and manages the NVM nodes independently from the DRAM nodes. It also provides on-demand page allocation and free space management support. Applications can allocate to NVM explicitly using library allocators for persistence (*nvmalloc()*), or non-persistent capacity (*npmalloc()*) use. In addition, pVM offers a set of NVM-specific NUMA policies such as *nvmrevert* and $nvmpreferred$ (discussed in Section 3.3) by extending the OS and adding support to the

existing user space NUMA libraries. Applications and allocators can directly map NVM memory with a *nvmmap()* interface. We describe the design and implementation details in Section 3.3.

For fast persistent storage, applications use a persistent library allocator similar to the ideas proposed in Mnemosyne [146], and NV-heaps [56]. Figure 8 shows the programming model for persisting an image object. pVM extends the VM subsystem by providing an object-based persistence that maps each persistent object to a set of NVM pages with ACID guarantees for both OS and application data. Every object is identified by an object identifier (objID) which is a combination of a globally unique identifier (gUID) per application plus the hash of an object name. Hence, applications sharing an object should use the gUID and objID as a capability, similar to a shared memory implementation in Linux. Prior research [146, 56] and other user space libraries [87] also use similar object naming schemes. Our current object store implementation provides basic functionality such as object creation, updates, deletes, renaming, write protection, history of object updates, logging and durability guarantees. For the object logging, we use the NVML implementation. Because most applications today use a file format, when using pVM, we use a simple file-to-object converter to convert input files to an object format.Finally, pVM achieves improved performance due to its design approach which extends the virtual memory subsystem and its cache-efficient data structures and mechanisms.

Limitations. pVM's object store, since it only supports a flat namespace, is a good fit for applications that are not reliant on the hierarchical filesystem. Examples of such applications include persistent key-value and photo stores, and NoSQL databases. Hence, **pVM is not a replacement for filesystems**, and does not support comprehensive filesystem security policies such as link/unlink, aliases, group permissions. pVM uses unique gUID and objUID identifiers to prevent object name collisions and is similar in approach to prior systems that use flat namespaces [146, 56]. In pVM, applications that share objects can implement their own concurrency and transactional mechanism. Our ongoing work focuses on addressing these challenges with a shared memory consistency protocol.

3.3 Design and implementation

The pVM system is composed of (1) a VM-based OS NVM manager (pVM-OS), and (2) a user-library (pVM-lib) that provides an interface for applications to access the NVM.

3.3.1 pVM-OS support

We first discuss the hardware abstraction of NVM from the view of an OS, followed by the OS-level extension required for supporting capacity and persistence. pVM provides a cleaner abstraction of NVM memory and more importantly improves the processor cache and TLB use by extending the VM subsystem. Note that the VM subsystem has undergone decades of cache and TLB optimizations, unlike the VFS subsystem that has mainly focused in improving the storage performance.

3.3.1.1 pVM-OS hardware abstraction

Using NVM as a NUMA node. pVM-OS treats NVM as another NUMA node, and extends the OS to support NVM-specific memory allocation and management policies. Figure 62 shows a high-level design. By using a node-based abstraction, pVM provides a cleaner hardware abstraction to the OS and the user space applications without requiring significant changes to the current system stack or the data placement mechanisms and tools, including user space NUMA libraries. With this node-based abstraction, pVM can easily extend the DRAM-based NUMA management and data placement policies to support NVM-specific policies. Importantly, this allows applications to scale seamlessly across DRAM and NVM nodes. pVM with its VM-based design can easily support additional features such as memory hot-plugging. Because byte-addressable NVMs are not commercially available, in order to evaluate the benefits of the VM-based design, we emulate NVM by treating a DRAM socket as an NVM node, and we add special flags to the node data structures to prevent general purpose allocation to the NVM node. Pages are allocated from NVM only when the pVM-OS manager requests them. Because pVM supports both capacity and persistence, it divides NVM into three regions (1) the capacity region, (2) the persistent storage region, and (3) the OS-level persistent metadata and log region for object stores. Each region

corresponds to a new Linux memory zone, and we discuss shortly how pages are allocated from each of these zones.

pVM NUMA policies. The benefits of providing fine-grained memory placement policies for the application, library, and OS in multi-socket NUMA platforms have been extensively evaluated, along with performance gains from increasing data locality. Support for flexible memory placement policies are important when NVM is used as a memory because NVMs have lower bandwidth and higher write latency, but larger capacity (2-4x more than DRAM). Prior VFS-based solutions that allow the use of NVM for capacity [67, 146, 56] do not provide such placement flexibility. In contrast, by extending the VM-subsystem and by using a NUMA-based NVM design, pVM can easily add such policies. For instance, we added three new policies – *nvmrevert*, *nvmpreferred*, and *nvmbind* – to the OS and the existing user-level Linux NUMA library. As the names suggest, *nvmrevert* reverts to using NVM when all DRAM pages are exhausted, *nvmpreferred* refers to using NVM as a default memory allocation node, and *nvmbind* restricts the application to only use NVM. We evaluate the benefits of such policies for memory capacity-intensive applications in Section 7.1.5.

3.3.1.2 pVM-OS software abstraction

While a NUMA-based configuration provides a cleaner hardware abstraction, it is important to extend the NVM OS data structures across different layers of the VM subsystem. Figure 10 shows multiple layers of virtual memory starting with the OS interface memory access layer at the user-level that provides the *mmap()* or *nvmmap()* interface, followed by the memory mapping layer that maps an NVM region into the user address space, and finally, the low-level allocator responsible for allocating an NVM page. We first review existing VM data structures and then discuss the changes required at each of the OS layers to enable NVM support.

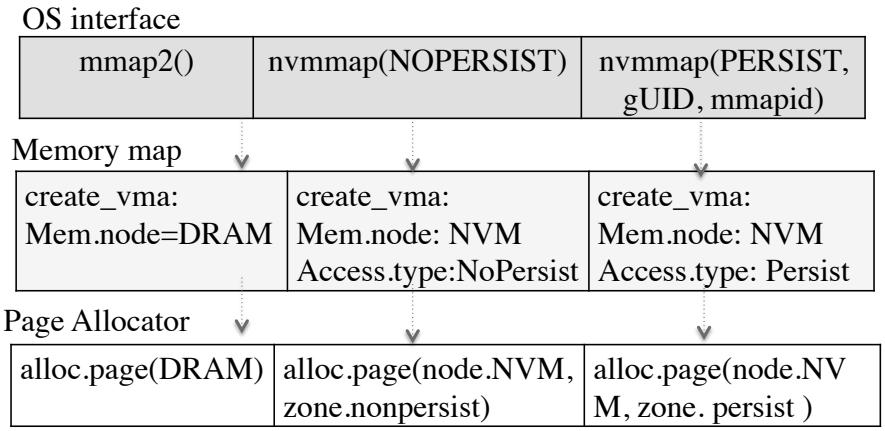


Figure 10: Extension of VM layers to support pVM.

VM background. In OSes such as Linux, each process has an OS context and an associated memory structure (*task mm_struct*) that encapsulates the entire user address space (heap, code, data and stack segments). A process address space can contain one or more contiguous memory regions known as virtual memory area regions (VMA). All pages in a VMA have the same access permissions, and they are used for the same purpose (heap, code pages, etc.). Note that the VMAs are either created and/or merged when applications map memory to their address space using *mmap()* or *sbrk()*. The pages are lazily added to a VMA only on the first touch (a minor page fault). These VM structures (process context, VMA, and pages) are also necessary for building the OS-level persistent state of an application.

Capacity and persistence OS interface. pVM exposes a *nvmmap()* interface to user space applications and libraries, which allows them to map an NVM region explicitly into an application's address space for capacity and persistence use, as shown in Figure 10. The system call signature is similar to the *mmap()* interface, except that the additional flags NOPERSIST and PERSIST are used to distinguish between the capacity and persistence uses, respectively. In addition, for persistence use, application and VMA naming arguments are supplied by the user-level object store library.

Mapping NVM into the address space. The memory mapping layer creates and updates the process memory and VMA structures. Therefore, for cleaner partitioning of NVM and DRAM data structures, the memory mapping layer is extended to distinguish between NVM and DRAM nonpersistent data structures and also between the capacity and persistence data structures. As shown in Figure 10, pVM introduces a special VMA memory type flag to differentiate between DRAM and NVM regions and an additional access type flag to further distinguish persistent from non-persistent NVM regions. In contrast, VFS-based NVM mechanisms do not differentiate between persistent and non-persistent NVM regions, thus imposing metadata bookkeeping penalties for both capacity and persistence page access, as shown in our evaluation (Section 7.1.5).

NVM page allocation and per-CPU list. The OS allocator is the heart of any VM-based management. OSes such as Linux use the buddy allocator that has undergone decades of research and optimization. pVM aims to reuse and exploit most of the logic and optimizations, without sacrificing the goal of cleaner separation between DRAM and NVM structures. Hence, we use the VMA memory type and access type to differentiate between page allocation requests, and redirect allocations to the persistent or nonpersistent NVM zones. For persistent allocation, the allocator is also responsible for checking whether the requested page already exists but is not mapped to a process address space, in which case it adds it to the page table. Furthermore, to reduce the overhead of allocator complexity, OS allocators maintain per-CPU free page lists as a fast path for allocation and reclamation, before requesting the buddy allocator. Currently, however, this is limited to homogeneous memory only. We extend the per-CPU lists with an array of per-CPU lists, containing a separate list for DRAM, capacity, and persistence allocations. Note that, after a page is allocated from a persistent list/zone, the allocator does not have control over the pages, and such pages are reserved, non-swappable, and managed by the OS persistence manager, until released by an application with appropriate permissions.

Configuring capacity and persistence size. For a user-space application to make the

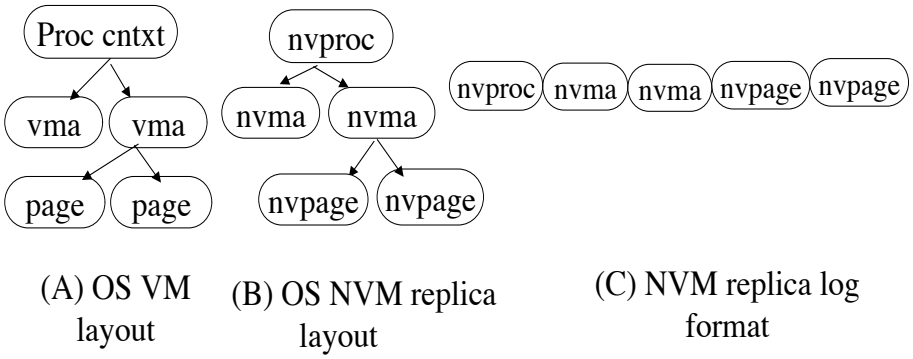


Figure 11: pVM-OS persistence structure layout.

best use of NVM, it has to know the space utilization of the capacity and persistence use, and the available options to configure them. Hence, pVM provides users with an option to reserve the capacity of persistent and non-persistent zones and provides dynamic information (via Linux' 'meminfo' tool). However, reserving the capacity does not require allocation and addition of pages to the page table. In the future, we plan to explore dynamic partitioning mechanisms.

3.3.1.3 pVM-OS persistence support

Providing support for persistent object stores requires OS and user-level library support. The OS is responsible for persisting all data structures such as process context, VMAs and pages that form a mapped region, whereas the persistence of the actual content, i.e., application data, is the responsibility of a user-level library. In other words, the OS provides metadata persistence for mapped regions, whereas the user library manages the region to offer applications a persistent object store. We first describe the OS support for the persistent mapped region, and then discuss the user-level object store support.

Complexity of persisting VM structures. When using the VFS, the filesystem and its metadata already support persistent regions. In pVM's VM-based design, three important OS structures that form a mapped region require persistence: the process context, all VMAs inside an application, and finally all page structures inside each VMA. However,

persisting these in-memory data structures is significantly complex. Each of these structures dereferences other OS structures, as they are continuously updated by different OS subsystems. Moreover, the VM data structures are designed for volatile DRAM, and hence are not persistence-friendly.

Key Idea: Replica log structures. To make the process context, VMA, and the page structure persistent, we propose a log-based approach, where we create simplified persistence-friendly replicas of these data structures. Each replica contains only the necessary information that is required to locate and load all persistent pages of a process. We term the replicas nvproc, nvma, and nvpage, corresponding to their in-memory process context, VMA, and page structures, respectively. Figure 11(A), shows the in-memory state of a process context with VMAs and related pages. Figure 11(B) in the middle shows the corresponding NVM replica structures (nvproc, nvma, nvpage) in a tree representation. Figure 11(C) represents the persistent storage log format of the replica structures.

Each process context replica, nvproc, can have a tree of nvmas, and each nvma can have a tree of nvpages. Each nvproc is identified by a globally unique identifier (gUID) supplied by the user library that maps a region of NVM using the OS interface *nvmmap()*. Each nvma contains the start and end physical addresses corresponding to the virtual address range of the VMA, and a locally unique VMAID supplied by the user library (generally an incremental *mmap()* counter). The VMAID is used for indexing the process nvma tree. Similarly, the nvpage has a physical address of the VM page and an offset set to the starting address of the VMA to which it is associated.

Creating and reloading persistent mapped regions. A persistent mapped region is created by an application or an object store library using the OS *nvmmap()* system call (see steps ① to ⑦ in Figure 12(A)). The library also provides a persistent map flag and a globally unique identifier (gUID) for each application. pVM's OS-level persistence manager uses the gUID to locate the persistent state of the replica process context in the log, and if it cannot find one, it creates a new context. Next, a VMA with appropriate persistent

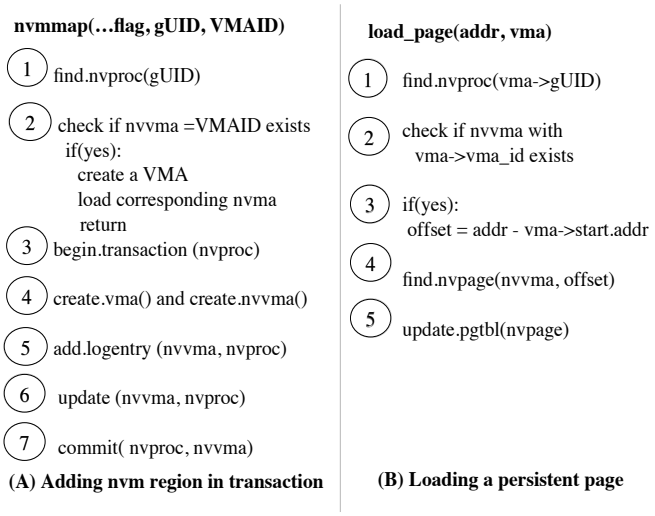


Figure 12: pVM persistent region creation, and page load.

flags is created along with its VM, and it is then added to the replica nvproc's nvma tree, as shown in Figure 11(B). For new persistent pages allocated and added to the in-memory VMA structure, a corresponding nvpage is created and added to the corresponding nvma RB-tree, indexed by the page offset.

To reload or read a persistent region (see Figure 12(B)), the application/user-library provides the gUID and the persistent region VMAID maintained in its user-level persistent metadata. The gUID and VMAID are used to locate the state of the corresponding regions in the persistent replica logs and load them into a tree structure. The pages are loaded into an application address space only after the application accesses a persistent page in the mapped persistent region. During the first touch, a page fault is generated, and by finding the offset of the faulting address with the starting address of a VMA, a corresponding nvpage structure is identified. This contains the physical address of the page to be loaded and added to applications page table. We favor this lazy approach as it reduces restart and read times and limits TLB pollution.

Consistency and durability. The OS layer is only responsible for maintaining consistency and durability of the OS-level persistent state, i.e., the replica structures required to construct the persistent region. The consistency and the durability of actual application

content are managed by the user-level object store. pVM borrows the ideas and code from PMFS's [67] optimized OS-level UNDO journaling with atomic commits for cacheline-sized updates. While PMFS maintains the consistency and durability of the filesystem metadata, pVM maintains consistency and durability for the VMA replica structures. Furthermore, a key difference between the PMFS filesystem and pVM design is that, rather than keeping a single journal of the entire filesystem, pVM maintains a separate journal for each gUID, or each process replica structure, with a global master journal bookkeeping the location of the individual process replica journals. Having multiple journals avoids contention for single journal lock across applications, specifically during frequent page updates. However, pVM currently requires applications to handle object sharing and to deal with concurrent updates. We plan to extend pVM with a more transparent mechanism, similar to the one presented in [152].

Figure 12(A) shows the high-level transactional journaling code for creating a new nvma structure. For updating an nvma, pVM first logs the nvma and its corresponding nvproc structure, and then logs the nvpage and nvma when adding or updating a new page. Each log entry in a journal is 64 bytes and the header and data of a log entry are committed first, followed by the log tail. Log updates are ordered with optimized write barriers (WB_BARRIER) and cache flush (CL_FLUSH) instructions discussed in detail by prior work [67]. Recovery happens by reloading first the master journal, followed by the log data in the journal. pVM follows an all-or-nothing model to guaranteeing ACID properties for OS persistent state. A failure to load any one persistent structure of an application makes an entire application state unusable.

3.3.2 pVM-lib allocator and object store

We next discuss pVM's user-level library that provides the application with a NVM allocator for capacity use and persistence object store support.

pVM capacity allocator and NUMA policies. Modern allocators map large regions of memory from the OS and manage them for subsequent allocations. For pVM, we extend the scalable jemalloc [1] library allocator with *nvmmap()* support for allocations to

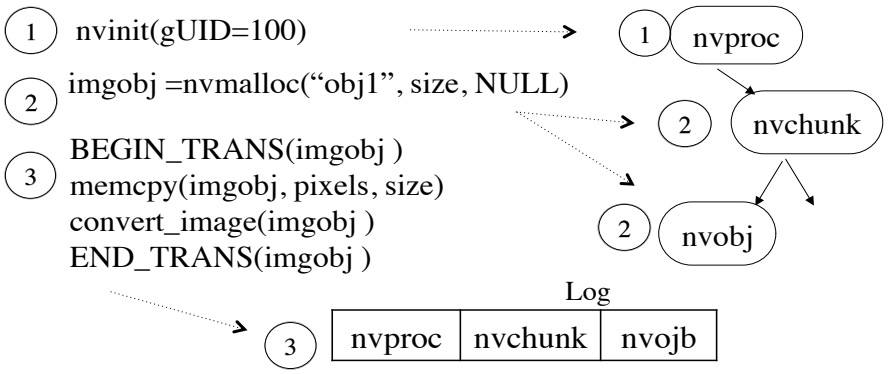


Figure 13: pVM object creation & updates.

NVM. We use jemalloc for both capacity and persistence allocation with appropriate flags to distinguish them. By using a library allocator, no application-level changes are required for nonpersistent capacity applications. Applications can also use the NVM NUMA policies.

pVM object store interface. pVM's object store interface and ACID mechanisms rely on named objects as inspired by prior NVM-as-heap research [56, 146]. We first briefly describe the application interface, followed by its object store metadata management customized for VM-based OS design. The object store provides applications with a *nvmalloc()* interface to create named, uniquely identified object (objID) formed by combining the per-application gUID and the unique object name. Figure 13 shows a sample code to allocate an image object and update it using the load and store interface. To provide consistency and durability, pVM uses Intel's persistent memory library [87] to wrap updates to an object inside transactions. We discuss the details next.

Persistent object store metadata. The persistent memory object store creates objects using a persistent memory allocator and maintains persistent metadata (state) about the objects, including their location in the mapped regions, size, objID, and information about consistency. Modern allocators are complex and maintaining the entire state of the allocators in the NVM can be expensive. Hence, pVM creates simplified replica structures in a log, similar to the OS-level replica structures discussed earlier. During application

initialization, a unique gUID is generated, and a corresponding user space nvproc structure is created in a user-level persistent log, as shown in step ① in Figure 13. Next, the allocator creates a large persistent region with a gUID, and an incrementing MMAPID (used as VMAID by the OS). Upon successful creation of an NVM region, a corresponding nvchunk structure is added to the NVM metadata, as shown in step ②. Each application (nvproc) can have several mapped regions (nvchunks), and for each object in the persistent region, a corresponding nvobject is added to the nvchunk. For consistency and durability, the objects and the object store metadata are updated within transactions, using an UNDO log ③. Note that the UNDO log is a journal located separately from the persistent metadata. Our consistency and durability mechanism is borrowed from object-based logs used in prior work [56]. For reading or recovering a persistent object, applications use the *nvread()* interface, with the gUID and object name. The object identifier generated from the combination of gUID and object name, is used to locate the corresponding persistent region and to instruct the pVM's OS manager to load the region into the application's address space. Furthermore, the consistency and durability of the persistent allocator is important. After a restart or recovery from a failure, the persistent allocator first recovers all nvproc, nvchunk, and nvobject data structures from the allocator log, rebuilds the allocator state, and garbage collects all uncommitted and unused objects [93].

Discussion. Several research proposals have focused on optimizing NVM object stores that rely on the VFS. However, pVM's main contribution is its generic VM-based OS (pVM-OS) design that addresses capacity bottlenecks and improves object storage performance. To understand the effort required to adapt other open source object stores to the pVM-OS design, we extended Intel's SNIA-based NVML library [87] with 75 lines of code. NVML maintains its own object store metadata. Hence, to create pVM-based persistent regions, we replaced the VFS-based *mmap()* with *nvmmap()*, letting us map the objects to the corresponding NVM regions using the unique ID for the mapped region similar to pVM's gUID.

Table 5: Applications for pVM evaluation.

Applications	Description	Workload	NVM usage type
FaceRec [19]	OpenCV face recognition on Gallagher dataset [71]	1.2GB input DB (2K images, each 800KB)	Capacity
Metis [44]	Uses Metis with 4 mappers-reducers	2GB crime data set	Capacity
GraphChi[96]	Graph Pagerank algorithm	Orkut graph, 117 million edges	Capacity
Dedup [41]	Parsec Dedup benchmark	4GB OS image file	Capacity
Snappy [23]	Fast data compression used in several Google products	Image, video, audio, document files -2GB	Object store
LevelDB [74]	Google's DB used from browser to datacenter	SQLite benchmark 500K operations	Object store
Phoenix [157]	Shared-memory MapReduce running word-count	same as Metis	Object store

3.4 Evaluation

We next evaluate the memory scaling capability of heterogeneous memory to : (1) understand their memory capacity scaling capability, (2) analyze the cache and TLB usage efficiency and page allocation and access time,

3.4.1 Methodology

Experimental Setup and NVM Emulation. For our evaluation, we use a 2.4 GHz, 8 core Intel Nehalem platform with 12MB LLC, dual NUMA socket with each socket containing 3GB DDR3 memory, and a Intel-510 Series SSD. pVM is based on the Linux 3.9 kernel, whereas for PMFS and ramFS we use Linux 3.11. Mnemosyne uses a legacy 2.6.33 kernel. We use one of the NUMA socket as an NVM node. To emulate NVMs with 10x lower bandwidth and 2x or 5x slower read/write speeds relative to DRAM, we first use thermal throttling of a NUMA socket to reduce the bandwidth. To emulate latency, we dynamically inject delays into the application runtime by finding the number of load/store LLC misses after a fixed interval of time (100 ms for our experiments). While Dulloor et al. [67] use a similar technique by modifying the processor microcode-level on a specialized server-class machine, we emulate this technique via software since we lack access to such microcode. For our experiments (capacity and object store), we consider five configurations, ① DRAM

Table 6: Comparison of approaches.

Approach	Memory scaling	Mgmt. subsystem	NUMA awareness	Consistency & durability
Mnemosyne	No	Filesystem	No	Yes
ramFS	Yes	Filesystem	No	No
PMFS	No	Filesystem	No	Yes
pVM	Yes	Virtual memory	Yes	Yes

only – a 6GB DRAM only configuration, ② PMFS – a split 3GB DRAM and 3GB PMFS managed approach, ③ Mnemosyne with 3GB DRAM and a 3GB NVM managed by the storage class memory map (scmmap) driver, ④ ramFS - a memory-based filesystem with 3GB DRAM and a 3GB ramFS filesystem that does not offer any ACID-based persistence guarantees (unlike PMFS), and ⑤ pVM - 3GB DRAM and a 3GB NUMA node managed by pVM. Table 6 compares the functionality and the design for each of the approaches.

Applications' use of NVM. We use a set of benchmarks (discussed along with the results) and applications that are shown in Table 14 to evaluate the capacity scaling and object store benefits and implications of pVM. In Table 14, we categorize the applications into two types based on their NVM usage (last column). Applications marked as 'capacity' use NVM only for memory capacity scaling and keep input data in the SSD. Applications marked as 'object store' use NVM for both persistence and additional capacity. For all approaches, (pVM, ramFS, PMFS, Mnemosyne) we use the jemalloc allocator [1] from Intel's NVML library [87] because of its multithreading scalability. PMFS and ramFS map memory from their respective persistent store using the MAP_PRIVATE flag, whereas Mnemosyne uses the MAP_SCM flag. For pVM, we replace the *mmap()* interface in jemalloc with the *nvmmap()* interface. For memory scaling analysis, we use pVM's Linux NUMA library extensions that support NVM-based NUMA policies. For persistent storage analysis, we use applications that are a good fit for object-based storage (LevelDB, Snappy, Phoenix), as discussed by Harter et al. [83]. When using the object interface, we replace the POSIX interface with NVM's heap-based object store. Modifying Snappy [23] required less than one man-day.

Limitations of memory-based filesystems. ramFS and Mnemosyne extend the Linux

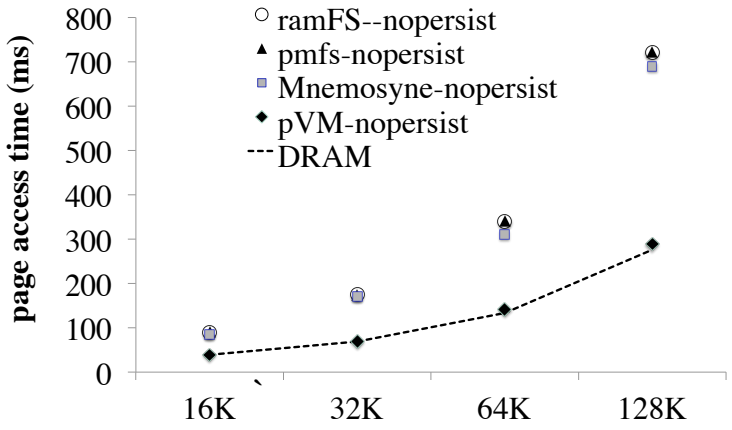


Figure 14: Non-persistent page access time for capacity use.

page cache to provide persistence using a simplified filesystem/persistence driver. Both of these approaches rely on an application-level allocator that first maps a large file with required flags. The flags and the file descriptors are used by the VFS-based component to allocate from any NUMA node that the current process is scheduled to. VFS uses the allocator managed by the VM subsystem, but capacity and persistence use are not differentiated. The allocated/used pages are marked dirty, which prevents the VM from recycling or swapping them to SSD. As a result, the application can non-deterministically terminate when memory usage exceeds the available free memory. We discuss these issues in more detail when evaluating the memory capacity scaling of applications.

3.4.2 Capacity use analysis

To understand the effectiveness of pVM's VM-based design for using NVM for additional capacity, we first use a set of benchmarks to evaluate page access cost and cache and TLB efficiency, and then we use real-world capacity-intensive applications to understand the implications of pVM's NUMA-based policies. We compare pVM with the other VFS-based approaches.

Page allocation cost is an important metric for understanding OS and application performance. Applications and allocators map regions of memory or NVM using the *mmap()* (or *nvmmap()* for pVM) interface for capacity and persistence needs. However, the actual allocation of a page happens only on its first touch. To measure the page allocation cost,

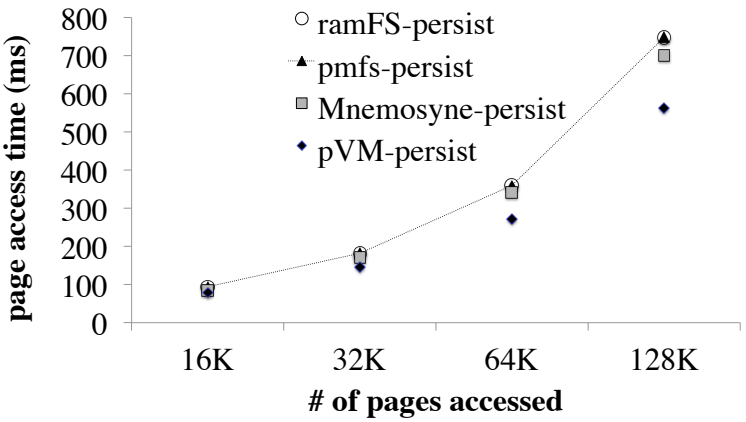


Figure 15: Persistent page access time.

we use the Linux memory mapping scalability benchmark [12], which first maps a large region of memory and then randomly touches pages across their page boundaries. Figures 14 and 15 show the performance of non-persistent and persistent page accesses. The x-axis represents the total pages accessed, and the y-axis represents the page access cost, including page allocation time and the time to update the page table. We analyze this for PMFS, ramFS, Mnemosyne, DRAM, and pVM.

Analysis. pVM distinguishes between non-persistent and persistent allocation using the flags provided to *nvmmap()*, and hence does not add filesystem metadata/bookkeeping or journal overheads for nonpersistent allocations. In contrast, the VFS-based PMFS lacks the ability to distinguish between capacity and persistence use and therefore adds filesystem overheads even for pmfs-nopersist access. Therefore, pVM-nopersist reduces the page access cost by 2.5x relative to pmfs-nopersist. ramFS and Mnemosyne also suffer from high page access costs, both similar to that of PMFS. Mnemosyne provides a heap-based interface to applications, but page allocation is handled by the VFS subsystem, which is implemented over the filesystem buffer cache and requires filesystem metadata access. We also observe that Mnemosyne provides less than a 10% benefit over PMFS, and benefits mainly result from avoiding the strict transactional updates of PMFS.

Persistent page access requires loading the filesystem B-tree or the dentry caches and updating several complex data structures such as an inode, an inode bitmap, blocks, and

Table 7: Parsec benchmark application characteristics.

App	Canneal	Facesim	Ferret	Stream cluster	Swaptions	x264
Type	Enginee--ring	Animation	Similarity Search	Data Mining	Financial Analysis	Media processing
WSS	2GB	256MB	128MB	256MB	512KB	16MB

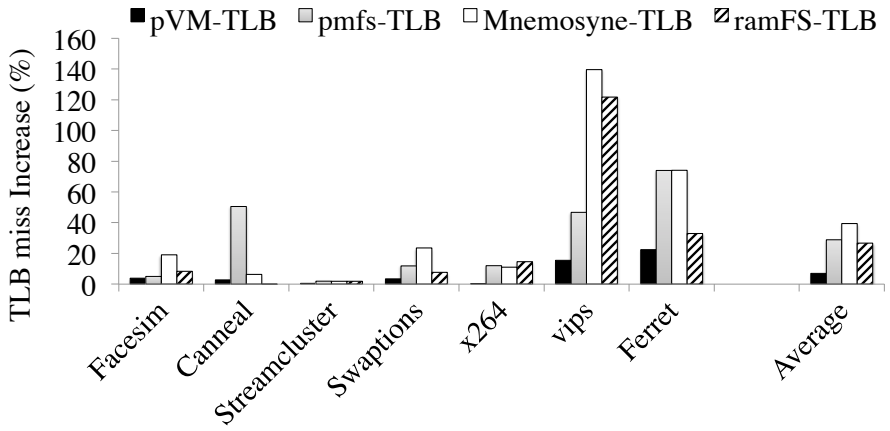


Figure 16: TLB miss analysis. Y-axis denotes TLB increase (%) relative to DRAM. The bars corresponding to 'Average' represent the average over all benchmarks.

a superblock. While pVM also has to locate pages from a VMA-level (nvma) RB-tree, only three simple replica structures, nvproc, nvma, and nvpage, are updated and journaled. Hence, pVM-persist shows a maximum performance gain of up to 20% compared to pmfs-persist and Mnemosyne and around 8% over ramFS, which does not include persistence guarantees. The small improvement of pmfs-nopersist over pmfs-persist comes from the use of the MAP_PRIVATE flags that instruct the filesystem to skip persistence operations on the page content. Using a B-tree instead of an RB-tree in pVM can improve these gains further. *This set of results shows the importance of using the VM-based design to distinguish between persistent and nonpersistent page allocation/access.*

We next look at how efficiently pVM uses the processor cache and TLB with applications from the well-known PARSEC benchmark suite [41]. We use a broad range of real-world applications from the suite (Table 7) with different working set size (WSS) and capacity

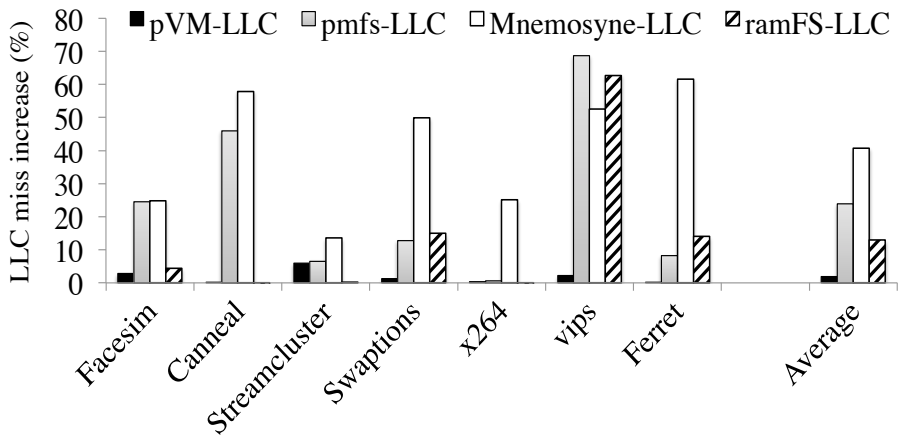


Figure 17: Cache (LLC) miss analysis. Y-axis denotes increase in (%) relative to DRAM.

requirements [40]. Figures 16 and 17 show the percentage increase in TLB and cache (LLC) misses (on the y-axis) relative to using only DRAM (baseline).

Analysis. As evident from the figures, pVM achieves a significant reduction in TLB and cache misses by extending the VM subsystem. It reduces the average TLB and LLC misses by up to 29% and 24%, respectively, when compared to the VFS-based PMFS approach, and even more when compared to Mnemosyne. By reusing most of Linux virtual memory management support, pVM incurs a negligible increase in TLB and LLC miss rates relative to the DRAM baseline. For highly memory intensive applications such as vips and ferret, the 10-22% increase for pVM is mainly due to higher remote node (NVM node) misses. In contrast, for PMFS, we notice substantially higher TLB miss rates (up to 60%), particularly when the working set size is large (facesim, canneal). A closer analysis reveals that these misses for PMFS are large because VFS does not distinguish between capacity and persistence pages, which adds transactional bookkeeping costs to the filesystem metadata. Note that filesystem data structures are not cache/TLB friendly when compared to VM-based data structures, which further increases the cache and TLB pollution for PMFS.

With Mnemosyne, we noticed a similar significant increase in both TLB and cache misses for memory intensive benchmarks while its VFS-based cache mechanism performed reasonably better for applications with lower memory intensity. Overall, the average TLB and LLC misses increased by 40% and 39%, respectively. For ramFS, the overheads were

relatively lower when compared to Mnemosyne. We attribute the difference to the following reasons: (1) Mnemosyne uses only an MAP_SCM flag and does not distinguish between persistent and non-persistent NVM use. Although both ramFS and Mnemosyne extend the VFS caching mechanism, Mnemosyne also provides persistence guarantees for a mapped region. These added overheads lead to a higher cost for persistence when compared to ramFS. (2) Another minor reason for the increase in TLB and cache miss rates is that Mnemosyne uses a legacy Linux kernel, 2.6.33, and does not incorporate the VFS and VM subsystem optimizations of newer Linux kernels. *These results demonstrate that a VM-based design leads to lower overheads with respect to TLB and cache misses.*

3.4.2.1 Memory scaling and placement impact

To evaluate the effectiveness of pVM's memory placement policies in addressing memory capacity scaling and performance issues, we use the applications in Table 14. For pVM, we evaluate two policies – 'pVM-nvmrevert' (revert to NVM if DRAM is exhausted) and 'pVM-nvmpreferred' (allocate first to NVM). The input data for the applications is placed in the SSD rather than in NVM, so that we can investigate only the capacity scaling benefits. As explained earlier, when using ramFS, PMFS and Mnemosyne for additional capacity, the user-level allocator has to map one or more large files to the NVM. However, the application's memory requirement can also sometimes exceed the available NVM space. Both ramFS and Mnemosyne use the VFS page cache that can dynamically scale over DRAM, whereas PMFS requires a reservation during boot. On the other hand, ramFS and Mnemosyne cannot release pages that have been allocated and written because they do not support swapping. Lack of swapping results in significant memory pressure, which can slow down the overall system and even cause termination of the applications.

Analysis. We first compare pVM against DRAM, PMFS, and ramFS in Figure 18. Clearly, DRAM (the optimal case) and pVM-nvmrevert outperform PMFS. In PMFS, when DRAM is exhausted, the OS lacks support to allocate transparently from a PMFS backed NVM because it uses an entirely different subsystem (VM versus VFS). With the increase in DRAM pressure, swapping is initiated, which results in the application slowing down. However,

pVM-nvmrevert provides a seamless memory scaling capability to switch automatically to NVM when DRAM is exhausted. While using NVM reduces performance compared to DRAM, it still provides a 2.5x and 2x speedup relative to PMFS for applications such as FaceRec and Metis. Interestingly, for Dedup with 3GB peak memory usage and 3GB NVM (a corner case), the PMFS approach (54 sec) marginally improves performance compared to pVM-nvmrevert (61 sec). This additional overhead occurs because pVM-nvmrevert starts allocating pages from the slow NVM after reaching a non-critical free page threshold for DRAM [76]. Normally, this logic prevents all DRAM pages from being exhausted and as such is also common in current Linux NUMA policies, which pVM builds on. In the PMFS case, it is not NUMA-aware, so it exhausts all 3GB without using slow NVM pages. A stricter, NVM-specific policy can be used to avoid these overheads. As expected, pVM-nvmpreferred has the highest cost. We expect the pVM-nvmpreferred policy to be useful only for low priority or memory bandwidth and latency-insensitive applications. pVM-nvmrevert achieves close to DRAM performance and significantly reduces cache and TLB (80%) miss rates versus the VFS-based approach (see Figure 7).

Unlike pVM, ramFS has no flexible policies. Hence, we map a large region of memory (6GB, which includes DRAM and NVM nodes) as ramFS and run the application with a DRAM preferred NUMA policy that would first allocate pages from DRAM and then from NVM. ramFS performs better than PMFS as it can dynamically allocate pages using the OS allocator, and it does not provide any journaling or ACID guarantees. As a result, for some applications, ramFS performance is similar to pVM (e.g., FaceRec and Graphchi). The performance of ramFS drops with memory capacity-intensive applications such as Metis and Dedup because dirty pages cannot be swapped, which results in the application slowing down due to memory pressure.

Figure 19 shows the comparison against Mnemosyne without the NVM latency and bandwidth emulation. Mnemosyne's legacy PCI driver does not export memory controller information required for DRAM throttling, so we just measure the software bottlenecks. As expected, Mnemosyne's performance trends are similar to those of ramFS, except with higher overhead for memory-intensive applications (Dedup and Metis), because it reuses

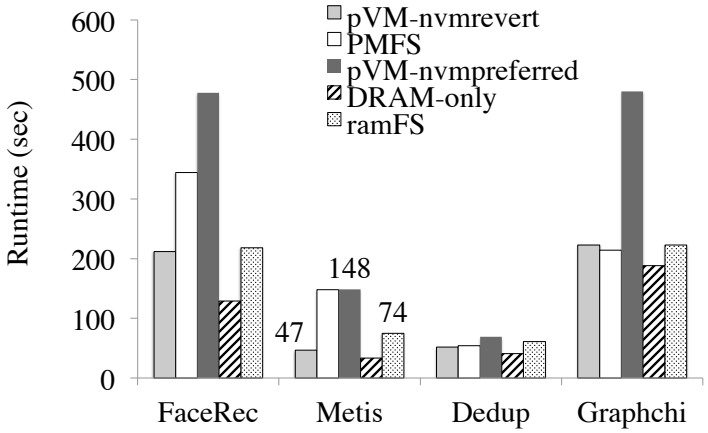


Figure 18: Memory scalability impact.

(pVM-nvmrevert, pVM-nvmpreferred are NUMA policies)

most of the ramFS page cache code. *pVM's memory scaling capability and flexible NVM memory placement policies allow for better performance with memory-intensive applications that scale across DRAM and NVM.*

3.4.2.2 Database benchmark

To analyze the impact of pVM's VM-based object storage performance, we first analyze the overall throughput and the OS-level overheads, such as journaling, for the popular NoSQL LevelDB key-value store [74, 75]. We evaluate: (1) pVM-obj, (2) pmfs-obj – pVM's user-lib for object store with memory mapped from VFS-based PMFS, (3) pmfs-mmap – a mmap instead of block-I/O version of the applications, (4) pmfs-block (baseline) – block-based file I/O with a traditional POSIX interface, and (5) ramFS – an approach with no consistency or durability guarantees. We do not evaluate Mnemosyne due to the lack of support for full NVM emulation.

LevelDB offers to applications a key-value interface and is a perfect fit for an object-based store. LevelDB internally uses a POSIX block interface for updates and a memory map interface for reads. Figure 20 compares the throughput for pVM and pmfs-obj relative to pmfs-block for 500K transactions of the LevelDB benchmark [75] where the x-axis shows the four types of access patterns. For Write-Sync, updates even within a transaction are committed (logged), unlike Write-Seq and Write-Random where commits happen only

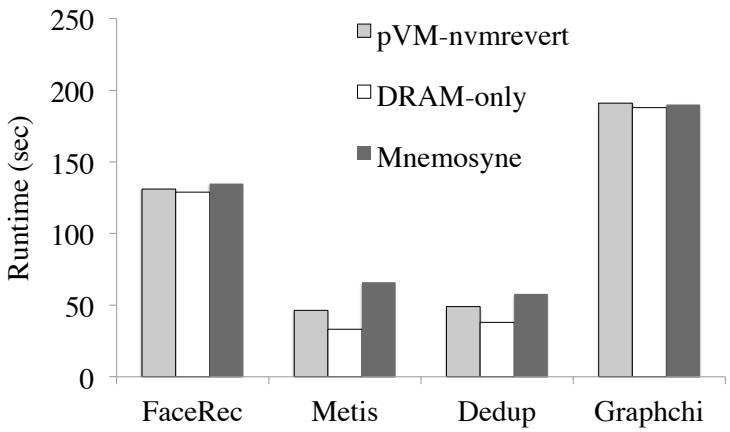


Figure 19: Memory scalability impact without NVM latency and bandwidth emulation.

outside the transaction. For read operations, the entire database is mapped to memory without significant performance differences between different approaches. Hence, we only study the write access patterns. Figure 21 shows the reduction in throughput (%) as a result of OS-journaling.

Analysis. Clearly, both pmfs-obj and pVM provide higher throughput relative to pmfs-block. pmfs-obj improves throughput for sequential and synchronous writes by 15-20%, respectively, whereas pVM-obj achieves a 30-53% increase. Both pmfs-obj and pVM-obj reduce the OS-filesystem and the user-kernel switch cost, unlike with pmfs-block and pmfs-mmap. Note that pmfs-mmap also requires several supporting I/O calls every time a file is mapped into memory. Other overheads include user-space-to-kernel data copy costs, and most importantly, higher OS journaling costs, as shown in Figure 21.

pVM, pmfs-obj comparison. Although pmfs-obj reduces the OS filesystem and context switching costs because objects are mapped to a region managed by the VFS (i.e., PMFS), we see higher OS-journaling overhead due to complex filesystem metadata, higher page access costs for frequent allocations and release operations for small transactions, as well as higher cache and TLB misses. Instead, pVM's persistent storage is specialized for object stores, and it needs to maintain a consistent state by using only its replica structures with some bookkeeping discussed in Section 3.3. As a result, the OS journaling cost with pVM

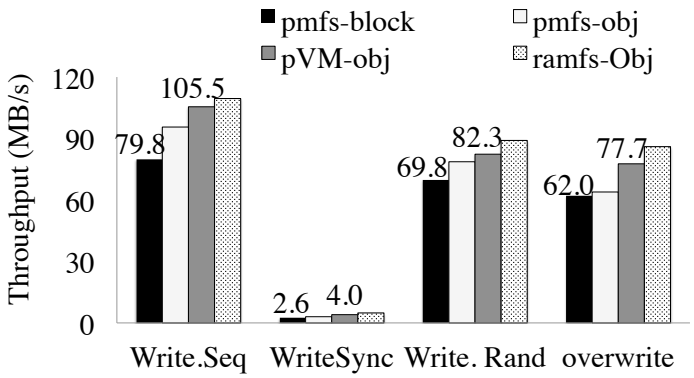


Figure 20: LevelDB throughput comparison (numbers over the bar in MB/sec).

Table 8: pVM cache, TLB miss, CPU instruction reduction relative to pmfs-obj.

App	Instruct (%)	TLB Miss(%)	Cache Miss(%)
Snappy	8.20	11.7	6.32
Leveldb	9.40	8.32	6.16
Phoenix	3.17	1.1	5.6

and its impact on database throughput are reduced, thereby providing better performance. ramFS, a hypothetical case without persistence guarantees, does not have an OS journaling cost or related cache and TLB misses. However, this approach is useful when an application does not require any durability guarantees.

3.4.2.3 Object store analysis for applications

Finally, we study the impact of pVMs VM-based design for improving the overall application performance and reducing the time spent in OS operations. We compare the functionally similar PMFS and pVM approaches.

Impact of object-based interface. In Figure 22, the left y-axis shows the file compression throughput for Snappy (MB/sec), and the right y-axis shows the time spent inside the OS and filesystem. Figure 23 displays the runtime comparison (left y-axis) for a word count in Phoenix with a 2GB data set, and the right y-axis shows the corresponding OS time. For Snappy, both pVM-obj and pmfs-object provide significant gains over the traditional

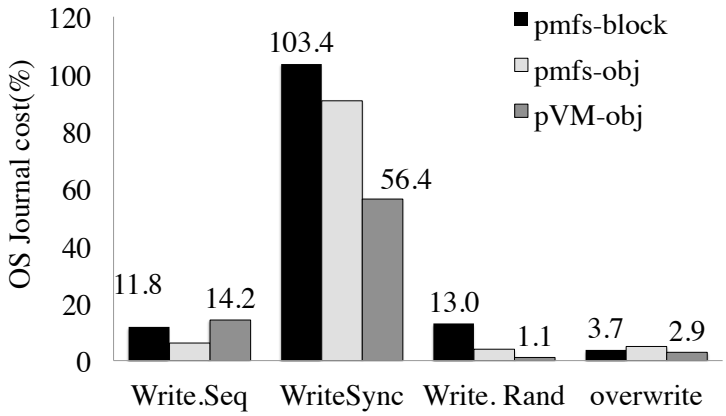


Figure 21: LevelDB: OS journaling cost (%) on throughput. ramFS does not support journaling.

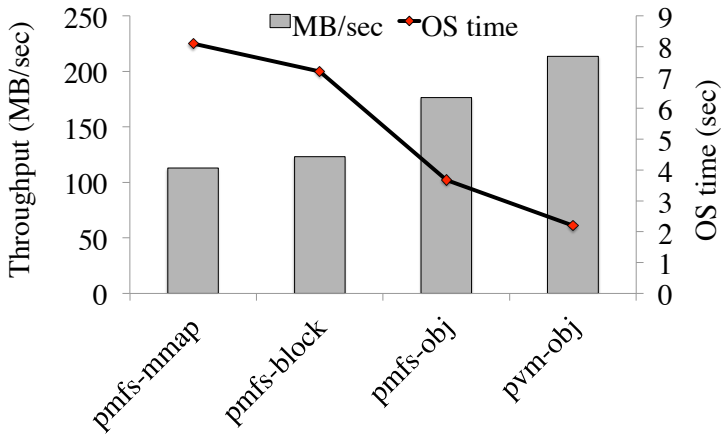


Figure 22: Snappy throughput, OS time.

pmfs-mmap and pmfs-block approaches. Applications like Snappy show high variability in I/O sizes and perform frequent I/O calls. Using an object-based interface (pVM-obj or pmfs-obj) significantly reduces the cost associated with context switches, filesystem meta-data, and the synchronization overheads of pmfs-block and pmfs-mmap. As a result, the time spent inside the OS (and specifically in the filesystem) is reduced by up to 4x with this interface. Interestingly, using the mmap interface with pmfs-block causes the performance to deteriorate when compared to pmfs-block. This performance difference is primarily due to the increased kernel mediation from several supporting system calls for files which have low processing time.

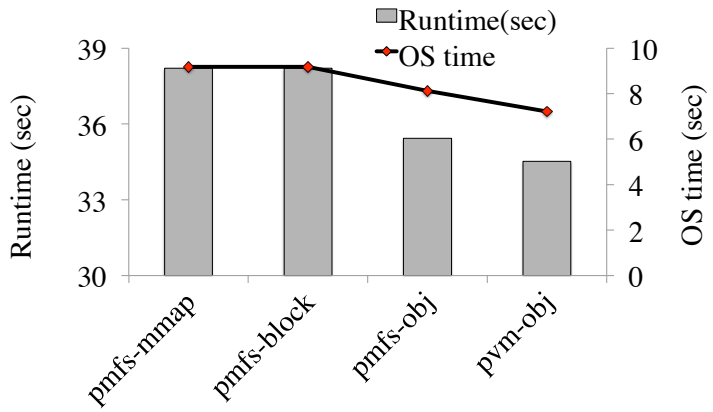


Figure 23: Phoenix runtime gains.

The second application, Phoenix MapReduce [157] maps the input data into the application address space, and then computes over the mapped input which is either backed by the VFS or pVM. Since no I/O calls are made after the initial mapping, the object-based interface provides less than an 11% improvement over pmfs-block. The memory map mode performs better compared to the block-mode by avoiding an extra copy from the filesystem to the DRAM buffer and has almost similar performance as pmfs-mmap.

Extending VM for persistence. When comparing pVM and pmfs-obj, pVM reduces the kernel time for Snappy by 67% relative to pmfs-obj, leading to an 18% improvement in throughput. Similarly, the throughput gains for Phoenix are around 14%. Although pmfs-obj uses pVM's object library at the user-level, the objects reside in files mapped and managed by the PMFS filesystem, and the PMFS page allocation component controls all persistent page allocations and accesses. In contrast, pVM benefits from faster page accesses that avoid filesystem-based metadata updates and that reduce journaling overhead cost, as discussed for the LevelDB benchmark. This results in fewer instructions and NVM accesses compared to pmfs-obj, as well as fewer TLB and cache misses, as shown in Table 8. Although the reductions are seemingly small, pVM's gains are of significant importance due to the 5x higher write latency of NVM relative to DRAM.

3.4.3 Summary of evaluations

We make the following conclusions from the experimental results presented in this section. pVM's VM-based design with NUMA capability addresses the memory capacity scaling issue by providing applications with flexible memory placement policies, and subsequently improves performance by up to 2x compared to PMFS (the pmfs-mmap and pmfs-block approaches). Next, by extending the OS to distinguish the NVM pages between capacity and persistence use, pVM reduces TLB and cache misses significantly – by up to 80% for the applications and 29% for the Parsec benchmark compared to the VFS-based PMFS. Furthermore, pVM's object store reduces software-stack overheads, page access time, CPU instructions, and cache and TLB misses, which results in nearly 2x higher throughput and up to a 4x reduction in OS time relative to block I/O. These results also motivate the need for OS redesign to better handle object storage.

3.5 Chapter summary

In this chapter, we discuss an OS-level solution, persistent virtual memory – pVM, for exploiting the capacity and storage benefits of byte-addressable NVMs. pVM integrates NVMs into the OS as memory (NUMA) nodes and extends the virtual memory to manage NVM nodes, instead of relying on the VFS to do so. In this manner, pVM addresses the memory capacity scaling issues of state-of-the-art designs and provides 'close to hardware performance'. pVM provides users with a flexible memory placement support which is integrated with existing user-level NUMA libraries. pVM further extends the VM subsystem for fast object-based storage to overcome substantial filesystem bottlenecks. By distinguishing between capacity and persistence access at the OS-level, apart from the performance benefits discussed in this dissertation, we believe there are several opportunities to bridge the gap between volatile memory use and persistent storage.

CHAPTER IV

SUPPORTING HETEROGENEOUS MEMORY IN VIRTUALIZED SYSTEMS

In the previous chapter, we discussed pVM, an OS-level NUMA-based design for capacity scaling and persistent object storage for heterogeneous memory. pVM provides an unified abstraction for the virtual memory to seamlessly scale across different memory technologies, however in heterogeneous memory, the central problem involves identifying performance critical memory regions and placing them in the fastest memory unlike traditional NUMA design where the software complexity is aimed at achieving data locality for the cores that access data. Most prior heterogeneous memory solutions have limited applicability because they rely on static analysis [35, 68] or require significant application-level changes [100, 67, 94] to identify critical memory regions and to migrate them to faster memory. Furthermore, these solutions lack a holistic system view, and therefore are not suitable for multi-tenant or virtualized environment. In virtualized datacenters, where hardware resources are shared across multiple virtual machines (VMs), the software management of the underlying memories can be complicated even more. In most current VMMs, guest-VMs are unaware of the underlying memory topology and lack fine-grained memory placement controls. A recent solution, HeteroVisor [79], tries to address this problem via a VMM-level, fully guest-VM transparent approach (referred to as VMM-exclusive hereafter) using reactive hotness-tracking and page migration. This approach suffers from significant hotness tracking and migration cost and fails to exploit the rich guest-OS information about the applications and their memory use.

4.1 Overview of HeteroOS

To address these problems, we propose **HeteroOS** – an OS design that transparently scales applications' memory capacity and manages memory placement in heterogeneous memory systems, leading to a significant performance improvement. HeteroOS derives its benefits

by first making the guest-OS heterogeneous memory-aware, and then extracts rich OS-level information about how applications use memory, such as pages used for heap allocations vs. pages used for I/O page cache. HeteroOS, associates *memory-usage*-specific policies with how pages are placed and migrated across the heterogeneous memory components. Although HeteroOS guests are heterogeneous memory-aware, they lack holistic view of the system required for resource sharing across multiple VMs, and do not have direct hardware control required for privileged operations such as hotness data tracking. To address this, HeteroOS also provides a novel guest-VMM coordinated management design – HeteroOS-coordinated, which permits the guest-OS to guide the VMM based on its deeper view of application-specific memory usage information. Finally, the HeteroOS design aims to explore generic principles for managing systems with a low latency, high bandwidth, limited capacity FastMem, and a low bandwidth, high latency, large capacity SlowMem. In this chapter, we,

- We propose HeteroOS, a design that makes guest OSes heterogeneity-aware by extending the existing NUMA-based OS memory management structures. Further, the guest-OS extracts information about applications' memory page use to provide smart memory placement that reduces migration.

- Guest-OS heterogeneous memory management is not sufficient since it lacks visibility and control in how hardware is really used. Hence, we propose a coordinated guest-OS–VMM approach that combines the OS-level management and application information to reduce the cost of VMM's privileged operations.

- Unlike prior research that focuses only on in-memory applications, we evaluate the HeteroOS design with memory, storage, and network-intensive cloud applications. Our design that includes guest-level heterogeneous memory management and guest-VMM coordinated management improves application performance by up to 3x relative to using only slow memory and by 2x compared to state-of-the-art VMM-exclusive management.

4.2 HeteroOS principles and design

Using the observations of the impact of heterogeneous memory on applications and the limitations of a reactive VMM-exclusive approach, we formulate the following principles.

4.2.1 Principles

Principle 1: Provide heterogeneous memory awareness to the guest-OS by extending the virtual memory.

To make the guest-OS heterogeneous memory-aware, the HeteroOS design retains the existing NUMA abstraction and extends the virtual memory subsystem. Important extensions include a new "on-demand allocation" driver that interacts with the VMM, changes to the page allocator, per-CPU and NUMA node data structures to make them heterogeneous memory-aware, and finally, a novel guest-OS HeteroOS-LRU page replacement to reduce FastMem contention.

Principle 2: Capture application's memory usage information from the OS subsystems.

Significant information can be extracted about how applications use memory from the pages allocated by different OS-level subsystems, (e.g., total heap pages for dynamic allocations vs. I/O page cache pages). HeteroOS captures such page-level information at the OS-level, and exploits the heterogeneity awareness to prioritize page placement to different memories without relying on expensive page migrations.

Principle 3: Enable a coordinated management to delegate and guide the VMM for performing privileged operations and resource management.

The VMM has a holistic view of the system resources and direct control to the hardware, but it lacks information about the applications running inside a guest. In contrast, the reverse is true for the guest-OS. Combining the capabilities of the guest-OS and the VMM can be powerful. Therefore, HeteroOS provides interfaces for coordinated management that enable the guest-OS to delegate to the VMM privileged operations such as hardware page table scans or updates for hotness-tracking, and heterogeneous memory resource management across multiple guest-VMs. The guest-OS uses its information about the applications'

memory use to guide the VMM what to track and when. The guest-OS also performs the page migration.

4.2.2 HeteroOS guest-OS design

The HeteroOS design includes (1) guest-OS heterogeneous memory awareness, (2) application-transparent memory placement by extracting OS-level information about applications' memory use, and (3) VMM-guest coordinated management. We discuss (1) and (2) in this section, followed by (3) in Section 4.3. Figure 24 shows the overall design with guest-OS and VMM-level components.

Heterogeneous memory abstraction as NUMA nodes. With HeteroOS, we aim to provide application-transparent heterogeneous memory support, while also leveraging most of the functionality currently provided by the virtual memory subsystem and its generic abstraction. Figure 24 shows a SlowMem and FastMem manager which are simple extensions to different parts of virtual memory to support heterogeneous memory. First, we expose heterogeneous memory (FastMem and SlowMem) as NUMA nodes with their corresponding guest-OS NUMA-related data structures. The initial capacities of different heterogeneous memory nodes are added to the guest's boot configuration. To enable the typically disabled NUMA data structures for a guest-VM, we make use of the Linux fake NUMA patch [134]. The software NUMA node structure has an additional one-byte flag to differentiate among different types of memories (e.g., FastMem or SlowMem). Additionally, guest-VM applications have flexibility to explicitly map memory to FastMem or SlowMem with an additional *mmap()* flag, but HeteroOS is not dependent on such application-level changes. Note that use of the generic NUMA node abstraction provides flexibility to extend this design to different memory technologies.

On-demand allocation driver. In current virtualized systems, the boot memory manager initializes the guest-VMs memory and adds a fixed number of pages under the control of the OS page allocator (free list of buddy allocator in the Linux OS). Additionally, for scaling up/down the capacity, the guest OS uses a balloon driver [149, 84], with its front-end in the guest-OS and the back-end in the VMM [149]. The pages allocated by the

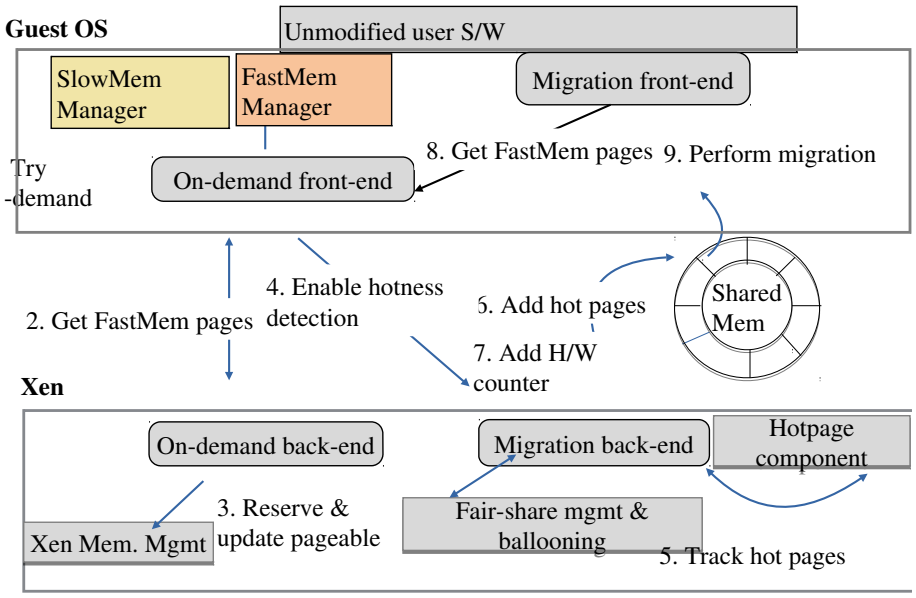


Figure 24: HeteroOS on-demand allocation, and coordinated management.

Steps 1-3 shows on-demand allocation, steps 4-9 shows hotness-based migration.

balloon driver are marked with a special flag and released to the VMM when the system memory pressure is high or during a garbage collection cycle. When exposing heterogeneous memory, the boot allocator should be able to initialize from different memory types, and the balloon allocator should be capable of scaling memory from different memory nodes. Using the Linux fake NUMA [134] enables multi-node memory initialization for the boot memory allocator. However, the balloon driver lacks such capability and requires significant changes. Hence, we add a new on-demand allocation driver. The driver's front-end in the guest-OS maintains multi-dimensional data structures required for memory-type-specific page allocation as shown in steps 1 and 2 in Figure 24. The back-end in the VMM handles the node-specific request and also maintains the per-node (memory type) machine page number (MFN) information for each of the guests. The front-end can also specify a fallback strategy when pages from a particular memory type cannot be provided.

Page allocation and per-CPU free list. First, new flags (FASTMEM, SLOWMEM) are added to page structure. These flags are useful for specializing allocation and replacement depending on the memory type. In Linux, the memory node is statically partitioned into three zones HIGHMEM_ZONES, NORMAL_ZONES, DMA_ZONES). Because of the

limited capacity of FastMem, all pages are allocated from HIGHMEM_ZONES (currently unused in 64-bit architecture) with no page reservation for other zones. Using the node abstraction reuses existing zones and avoids the complexity of introducing new zones for different memory types unlike other research such as [146, 154]). Because FastMem capacity is limited, we prevent page allocations that are not initiated by the HeteroOS page allocation manager.

Next, OSes such as Linux use a per-CPU free page list as a fast path for allocation and reclamation, reducing the use of the 'Buddy' allocator. However, currently, this is limited to homogeneous memory only. In HeteroOS, we replace the per-CPU list with a multi-dimensional per-CPU list to support different memory types. Because FastMem capacity is limited, most allocations are satisfied from free list which significantly boosts the allocation performance.

4.2.3 Memory placement and management

It is important to place the critical data structures of an application to capacity limited FastMem and others to SlowMem. However, the OS lacks information about the application-level data structures.

Key idea. HeteroOS exploits the guest-OS heterogeneity awareness and first extracts the memory use information of an application by categorizing pages as heap, I/O page cache, slab cache (buffer cache) based on the subsystem that allocated the memory page. I/O cache pages are traditionally used as a shield against a substantially slower storage and network medium, and the choice of placing them in FastMem versus SlowMem can significantly impact application performance. Unlike the current NUMA-based systems that only prioritize heap pages, HeteroOS prioritizes the cache pages when considering their placement to FastMem. Further, it employs a novel HeteroOS-LRU mechanism for resolving contention.

Heap allocation (Heap-OD). Most in-memory applications spend 70-80% of its time accessing its heap memory [68]. Therefore, placing heap pages to FastMem is important. Note that, as shown in Table 4, several large-scale applications (e.g., Graphchi) frequently

allocate and release pages. Providing on-demand heap allocation supported by a fast per-CPU freelist can significantly improve application speedup (up to 80% in our result).

Prioritize I/O page cache (Heap-IO-OD). The page cache plays a crucial role in accelerating the throughput of several I/O-intensive datacenter applications such as databases, graph analytics [96, 135, 4]. This is because the page cache increases spatial locality by reading ahead I/O pages and temporal locality by buffering dirty and modified blocks. Hence, we modify the filesystem page cache layer to allocate memory from FastMem when available.

OS kernel buffers (Heap-IO-Slab-OD). I/O-intensive (storage and network) applications frequently allocate kernel buffers from the slab allocator. As shown in Figure 4, for the network-intensive applications such as Redis, the allocations are dominated by the 'skbuff' for copying and sending user-space data or receiving packets at the device layer. For storage-intensive applications such as LevelDB, most I/O time is spent accessing the filesystem metadata stored in the kernel buffer. Note that kernel buffers are allocated only when required and immediately released after an I/O is completed, hence providing an opportunity for higher FastMem page reuse. Hence, to accelerate the I/O applications, we modify the slab allocator to use FastMem pages when available.

4.2.3.1 Resolving contention with HeteroOS-LRU

Key Idea. Prioritizing heap, I/O page cache and buffer cache (slab pages) allocations to a capacity limited FastMem can lead to capacity contention. However, guest-OSes already implement some form of LRU-based lazy page replacement mechanism that periodically swaps pages from (fast) DRAM to (slow) disk when the memory pressure is high. HeteroOS exploits its heterogeneous memory awareness and extends this lazy disk-based LRU to an aggressive heterogeneous memory-specific LRU.

Page replacement background and drawbacks. Linux uses a 'split approximate LRU mechanism' for page replacement and swap management. Briefly, the approach maintains two lists for each memory zone – an active list of hot pages, and an inactive list with cold pages. Each page has two bits, PG_active, PG_referenced to identify whether the page is

in the active list and referenced recently. Under memory pressure, pages on the inactive list are swapped to disk, and pages with less than two references are demoted from the active to the inactive list. At a high-level, this can be used for addressing the FastMem contention by placing active pages to FastMem and inactive pages to SlowMem, instead of using expensive VMM-level tracking.

First, in current NUMA systems, the LRU mechanism treats memory pressure as the overall system memory pressure (FastMem + SlowMem capacity) rather than memory pressure for each node (FastMem or SlowMem). Hence, the LRU active-inactive classification and the replacement do not get initiated as they do not differentiate between FastMem and SlowMem. Second, using only a lazy threshold-based eviction for LRU is insufficient for a limited capacity FastMem used by all the OS subsystems. FastMem allocations can increase, and importantly trigger a storm of evictions resulting in an application slowdown. To address these issues we design HeteroOS-LRU.

Solution - HeteroOS-LRU. First, HeteroOS extends the guest-OS page-replacement with support for memory type-specific thresholds for triggering replacement. Second, unlike the lazy approach, HeteroOS-LRU actively monitors the active to inactive state change of heap, I/O page cache, and slab pages and immediately evicts them from FastMem. Continuous monitoring is not feasible without free vCPUs. To further increase FastMem allocations and reduce misses, HeteroOS uses the following optimizations based on the memory type-specific threshold.

- Memory unmap and release: During an unmap operation, several continuous pages in a VMA region are released. HeteroOS-LRU marks these pages inactive and aggressively migrates them to SlowMem
- I/O page and buffer cache pages: These pages are released after an I/O request, are marked inactive and immediately evicted from FastMem.

4.3 Coordinated management

When the on-demand allocation and LRU mechanism is not sufficient to locate free FastMem pages, the guest-OS delegates the page hotness-tracking to the VMM and also guides it to

track only relevant pages. However, the actual migrations are performed at the guest-OS – a fundamental difference compared to the VMM-exclusive HeteroVisor approach as discussed in Chapter 2. The VMM also performs system-wide heterogeneous memory resource sharing across guest-VMs.

4.3.1 HeteroOS-coordinated design

HeteroOS's coordinated management (HeteroOS-coordinated) reuses HeteroVisor's hotness-tracking implementation, but modifies it substantially to support new guest-OS interface for coordination with the VMM via a split guest-OS front-end and VMM back-end model as shown in the steps 3-7 in Figure 24. We next discuss the design details.

4.3.1.1 Fast communication with shared memory

Coordinated management requires significant information transfers. This includes page hotness information, hardware counters, and memory usage information, transferred from the VMM to the guest, as well as guest-OS hints to the VMM for reducing the hotness-tracking cost (see Figure 24). To enable this, HeteroOS creates a shared memory channel between the guest and the VMM by reserving 1K FastMem pages when booting a guest-VM. The shared memory can be dynamically adjusted. The communication between the guest and VMM uses a customized record structure that contains a header (record type), a data portion (e.g., list of hot pages), and an 8-byte flag reserved for shared memory synchronization.

4.3.1.2 OS-guided VMM-level hotness-tracking

Tracking the entire guest-VM's memory for hotness is expensive. Hence, our coordinated management reduces the scope and cost by using the guest-OS information to guide the VMM about what pages to track, and when to track them.

Guiding what to track using OS-level information The guest-OS exports a *tracking list* and an *exception list* to the VMM using the shared memory channel. The *tracking list* contains address range of contiguous memory regions of a particular application of interest that the VMM should track for hotness. In Linux, this information can be extracted using

the virtual memory area (VMA) structure range [76]. Next, tracking short-lived I/O page cache and buffer cache pages only adds additional overhead. Hence, all such pages are to the exception list, and HeteroOS-LRU aggressively evicts them after the I/O request. Finally, in OSes such as Linux, kernel pages such as page table page or DMA pages are linearly mapped. Migrating them is complicated and is currently not supported by the Linux OS. Also, we observed negligible impact even when directly placing them to FastMem. Hence, we add the contiguous range of linearly mapped kernel and DMA pages (from DMA zone) to the exception list.

Guiding when to track using architectural hints. For applications phases with high page reuse and lower processor cache misses, tracking and migrating pages to FastMem will have limited improvement, give the cost of migration. However, with current hardware support, the OS estimates the application's memory/page use pattern by forcefully setting and resetting the PTE access bit. Consequently, a frequently accessed but high cache reuse page suffers from repeated migration. Although recent hardware proposals assume additional logic to capture page hotness in the memory controller (for accesses that miss the cache) [107], there is no hardware support for mapping cache misses to individual pages. To address this, HeteroOS periodically reads the LLC misses using the hardware counters exported by the VMM, and when the cache misses are high, the hotness-tracking and migration interval is reduced, and vice versa when cache misses are low. The simple but effective model shown in Equation. 1 significantly reduces tracking cost. $i, i-1$ in the equation represent the current and previous interval. This information is exported to the guest-OS via shared memory.

$$\begin{aligned} \Delta LLCMiss &= (LLCMiss_i - LLCMiss_{i-1})/LLCMiss_{i-1} \\ Interval &= Interval - (\Delta LLCMiss * Interval) \end{aligned} \tag{1}$$

Guest-OS-controlled migration. In HeteroOS, the VMM's hotness-tracking is exported to the guest OS, and the page migrations are performed by the guest and not by the VMM for the following reasons. (1) *Page state*: Before a page is migrated, the OS performs several checks of the source and destination pages to establish if the page is valid, not dirty, not reserved, and not mapped to the kernel or DMA region. Migrating a page

1: R = $\{r_1, ..., r_m\}$ ▷ total memory capacities
2: C = $\{c_1, ..., c_m\}$ ▷ used memory capacities initially 0
3: $s_i (i = 1..n)$ ▷ guest i dominant shares, initially 0
4: $VM_i = \{vm_{i,1}, ..., vm_{i,m}\} (i = 1...n)$ ▷ memory resources given to guest i
5: pick guest i with lowest dominant share s_i in queue
6: D_i - guest i's memory allocation request
7: **if** $C + D_i \leqslant R$ **then**
8:
end
$C = C + D_i$ ▷ update consumed vector
9: $VM_i = VM_i + D_i$ ▷ update $i's$ allocation vector
10: $s_i = max_{j=1}^{m}\{vm_{i,j}/r_j\}$ **else**
11:
end
return ▷ no more free memory

Algorithm 1: HeteroOS DRF algorithm

violating these conditions can result in an OS/application errors. Unlike the VMM, the OS virtual memory has all such information readily available, avoiding errors. (2) *Scalability via adaptive migration:* The VMM-level tracking overhead increases with increase in the working set size and the number of VMs. Guest-level migration provides the flexibility to use the application information at the OS-level to selectively migrate performance critical pages only.

4.3.2 VMM-level resource management

The VMM plays a critical role in managing system resources across multiple VMs. Specifically for memory as a resource, most VMMs provide a simple max-min fairness, where first, resources are allocated in the order of increasing demand, second, each VM receives its basic share of memory which it paid for, third, any unused memory is evenly distributed among users demanding more than fair share (over commit), and finally, the additional memory allocated to a guest-VM is reclaimed using well-known ballooning. However, there are two drawbacks: (1) the current ballooning mechanism works only for single-level of memory, and (2) the notion of max-min fairness is not effective when there are multiple resources to share.

Extending ballooning. To address (1), we first extend the Xen balloon driver [149] to support reservation and overcommit of multiple memory types. Next, in Xen, guest-VMs can specify a minimum memory that is allocated during boot, and a maximum memory that

can be dynamically allocated when memory is not over-committed. Hence, with HeteroOS, guest-VMs can specify their FastMem and SlowMem minimum and maximum – a capability that cloud providers can directly tie to a cost model, which is beyond the scope of this dissertation. When a guest runs out its minimum FastMem or SlowMem, the balloon driver increases the share by requesting the VMM which activates ballooning in other VMs to release free pages of the corresponding type. Note that the HeteroOS balloon drivers use the HeteroOS-LRU mechanism to find inactive pages, and when not sufficient, swap pages to the disk.

Weighted dominant resource fairness. Next, to address the limitations of single resource min-max fairness, in HeteroOS, we treat each memory type as a resource and extend the 'Dominant Resource Fairness' algorithm proposed by Ghodsi et al. [72] that generalizes the max-min fairness for multiple resources. The dominant resource fairness (referred as **DRF** hereafter) DRF first computes the share of each resource allocated to a guest-VM. The maximum among all shares for a guest-VM is its dominant share, and the resource corresponding to the dominant share is called the dominant resource as shown in Algorithm 1. There can be one or more allocation request to same resource. DRF prioritizes them in order of smallest dominant share value. Each guest-VM specifies a maximum resource allocation vector ¡FastMem.pages, SlowMem.pages¿ when booting, and the DRF attempts to maximize the fairness across multiple VMs for multiple memory types. Given the limited FastMem capacity, most VMs will have SlowMem as the dominant resource. To address this drawback, we assign weights when calculating the dominant share with resource vector specified as ¡FastMem.weight * FastMem.pages, SlowMem.Weight * SlowMem.pages¿. We currently use static weights ('1' for SlowMem, and '2' for FastMem), and plan to add dynamic weight estimation as a part of our future work.

4.4 Evaluation of virtualized systems

We next present the results of the experimental evaluation of HeteroOS using micro-benchmarks and real-world cloud applications. The evaluations aim to answer the following questions:

Table 9: Experimental setup.

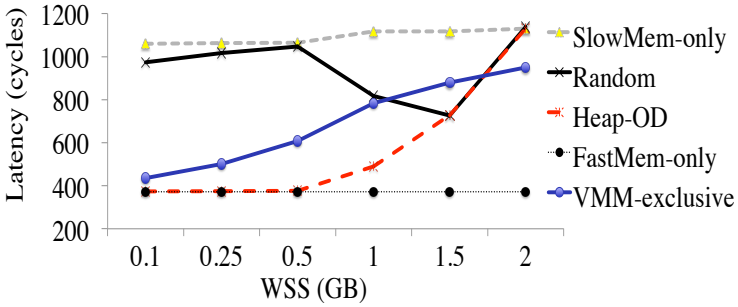
Mem. Nodes	2
CPU	Intel Xeon 2.67 GHz
No.of CPU cores	16
SlowMem capacity	8 GB per-VM
FastMem capacity	256MB-4GB per-VM
FastMem BW, lat	24.2 GB/sec, 66ns
SlowMem BW, lat	2.5 GB/sec, 320ns
Storage	Intel-SSD 320 series

Table 10: Summary of HeteroOS mechanisms. From top to bottom, each mechanism is incremental of the previous.

HeteroOS mechanism	Description
Heap-OD	on-demand heap allocation
Heap-IO-OD	Heap-OD + IO page cache allocation
Heap-IO-Slab-OD	Heap-IO-OD + NW buffer + slab allocation
HeteroOS-LRU	Heap-IO-Slab-OD + HeteroOS-LRU replacement
HeteroOS-coordinated	HeteroOS-LRU + OS guided hotness tracking + architecture hints

- What are the implications of making the guest-OS heterogeneous memory-aware?
- How effective is the proposed guest-OS memory placement for heap memory, storage, and network-intensive applications?
- What are the performance benefits of the coordinated guest-OS-VMM management when the FastMem capacity is limited?
- How effective are the heterogeneous memory's resource sharing mechanisms across multiple VMs?

Methodology. Table 9 summarizes the experimental setup for our evaluations. Future memory technologies such as DRAM, PCM (NVM) (and also Stacked 3D) are expected differ ∼10x in bandwidth with around 2-5x difference in latency as shown in Table 1. Hence, we emulate HeteroOS via DRAM memory throttling discussed earlier in Chapter 2 and [79, 59], and our emulation achieves ∼10x difference in bandwidth between FastMem (24.2 GB/sec) and SlowMem (2.5 GB/sec) and ∼5x difference in latency.



Miss Ratio	WSS (GB)	0.25	0.5	1	1.5	2
	Random	1.0	1.0	0.92	0.68	0.76
	Heap-OD	0.0	0.0	0.54	0.62	0.76
	VMM-exclusive	0	0	0	0	0

Figure 25: Memlat and FastMem Miss ratio

Applications and baselines. Table 14 shows the list of well-known real-world resource-intensive (CPU, memory, network, and storage) cloud applications. We use two baselines, (1) a SlowMem-only naive approach in which applications use only the SlowMem, (2) FastMem-only – an ideal case in which applications use only FastMem with unlimited capacity. Table 10 summarizes the incremental HeteroOS approaches.

4.4.1 Micro benchmark analysis

We use the 'memlat' [25] (Figure 25) and 'Stream' (Figure 26) benchmarks to analyze the implications of heterogeneous memory awareness on the memory latency and bandwidth from the guest OS' perspective. We limit the FastMem capacity to 0.5GB and SlowMem to 3.5GB. We compare (1) the VMM-exclusive approach that fully relies on page migrations, (2) a Random [52] approach in which the VMM adds 0.5GB of FastMem to a guest-OS that is not heterogeneous memory-aware, and randomly allocates FastMem and SlowMem pages, (3) Heap-OD, a HeteroOS guest which places all its heap to FastMem, and finally, the baselines (4) SlowMem-only and (5) FastMem-only. For the memory latency benchmark (Figure 25), we vary the working set size (WSS) from 0.25GB to 2GB in the x-axis, and the y-axis shows the measured average memory latency. The table in Figure 25 shows the FastMem page allocation miss ratio for each of the approaches. In Figure 26, the x-axis shows the Stream benchmark WSS (0.5 and 1.5 GB), and the y-axis shows the average measured software bandwidth.

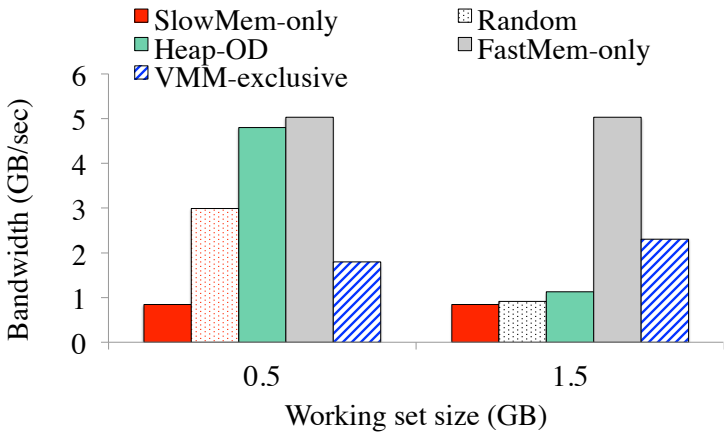


Figure 26: Stream benchmark.

Observation: First, when using the random approach, both the latency and the bandwidth benchmarks show non-deterministic behavior because of the random FastMem and SlowMem allocation. Hence, for a WSS smaller than the available FastMem (WSS of 0.25GB), the latency is high. Further increasing the WSS (0.5 and 1 GB), increases the memory pressure, resulting in a larger proportion of FastMem page allocations and lower latency. However, further increase in the WSS increases latency and decreases bandwidth. Next, with Heap-OD, when the WSS is smaller than FastMem, on-demand allocation results in an ideal latency and bandwidth comparable to the FastMem-only approach. Increasing the WSS beyond 0.5GB results in a gradual increase in latency and decrease in bandwidth. Finally, the VMM-exclusive approach with expensive hotness-tracking and page migration suffers from highest latency and lowest bandwidth, even when the WSS is small. However, for WSS even beyond FastMem capacity, VMM-exclusive is always able to allocate a FastMem page by using hot page tracking, and evicting and migrating a cold page to SlowMem and hence, the FastMem allocation miss ratio is always 0. Therefore, VMM-exclusive has a lower latency and higher bandwidth compared to Heap-OD for large WSS. The results highlight two important facts. *(1) On-demand allocation is important when the WSS is less than FastMem capacity, (2) for large WSS, hotness tracking, and migration is essential.*

4.4.2 Guest-OS memory placement

We next evaluate the effectiveness of HeteroOS in leveraging application page use information for right memory placement inside the guest-OS. Figure 27 shows the results for all applications listed in Table 14. We compare the following approaches summarized in Table 10: (1) Heap-OD, (2) Heap-IO-OD, (3) Heap-IO-Slab-OD, (4) HeteroOS-LRU, and finally, (5) FastMem-only - shown with the dashed line. Because the approaches are incremental, they are represented in stacks. We do not discuss NGinx [16] because it shows less than 10% heterogeneous memory impact. The y-axis shows the performance gain percentage compared to the naive approach of using SlowMem-only. In Figure 28, we show the miss ratio – a ratio of total FastMem page allocation misses to total page allocation request, used as a metric to measure the effectiveness of each approach in allocating FastMem pages. **Heap-OD:** First, even with simple heap-only placement (solid gray bars in the figure), heap-intensive applications such as Graphchi and Metis benefit significantly because of their time spent on accessing the heap data. Graphchi and Metis show 121% and 84% gains respectively with 1/2 capacity ratio. Metis with an active WSS of 5GB seldom releases pages, unlike Graphchi that frequently releases memory. Hence, for smaller FastMem capacity (1/8 and 1/16 ratio), Metis experiences a considerable slowdown compared to Graphchi. In contrast, Redis, an in-memory key-value store, is not only heap-intensive but also network buffer cache intensive, whereas as X-Stream and LevelDB are I/O page cache intensive. Therefore, Heap-OD alone provides limited gains in these applications.

Impact of Heap-IO-OD and Heap-IO-Slab-OD: As expected, placing the I/O page cache in FastMem provides substantial benefits for X-Stream and LevelDB (stripped bars in Figure 4). For LevelDB, buffering the database and its logs in FastMem speeds up read transaction with 2x improvement in throughput. X-Stream computes on the memory mapped I/O data, and hence, mapping the I/O to page cache almost doubles the Heap-OD gains. Graphchi, apart from being capacity-intensive, is also I/O-intensive, as it reads a large graph dataset to memory and frequently writes intermediate data as shards to disk. Placing the page cache in FastMem improves speed up, but the gains reduce when the FastMem capacity is reduced to 1/4 or 1/8 ratio, mainly from contention between heap

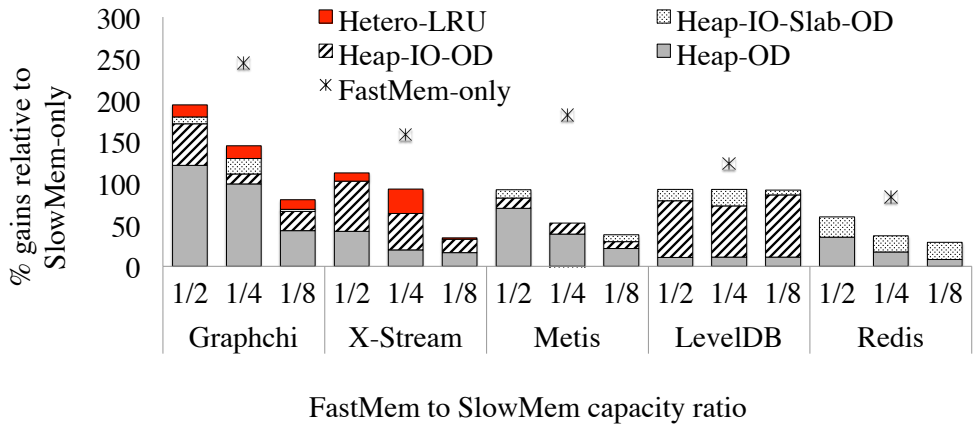


Figure 27: Impact of OS heterogeneity awareness. Y-axis shows gains (%) relative to using only SlowMem.

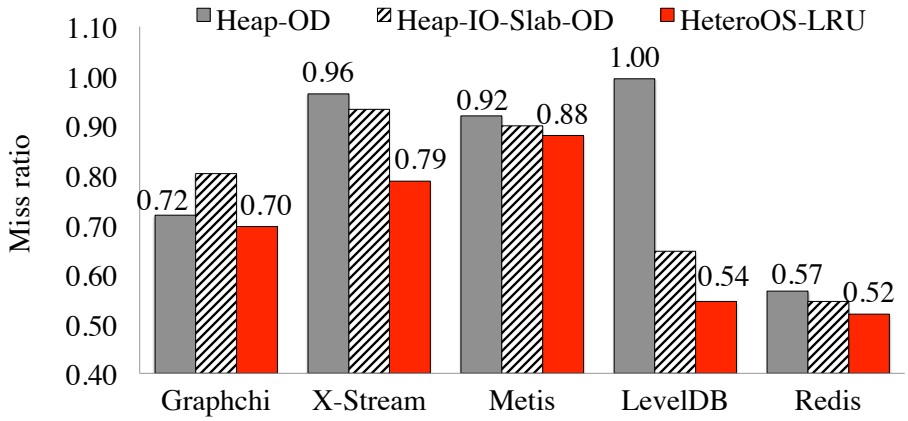


Figure 28: FastMem allocation miss ratio.

and I/O page cache pages. For the network-intensive Redis, placing the network send and receive buffers with slab allocation (Heap-IO-Slab-OD) in FastMem improves throughput.

HeteroOS-LRU: This mechanism reduces the contention between the heap and I/O cache pages by aggressively evicting from FastMem pages that are inactive or that are released after an I/O request. As Figure 28 shows, HeteroOS-LRU reduces the FastMem miss ratio for most applications. At 1/4 FastMem capacity ratio, HeteroOS-LRU improves Graphchi and X-Stream by additional close to 20-50% over Heap-IO-OD. For lower FastMem capacity ratios (1/8, 1/16), HeteroOS-LRU gain are limited (∼3-4%), and requires a more extensive hotness tracking and migration approach. *The results show the importance of heterogeneous*

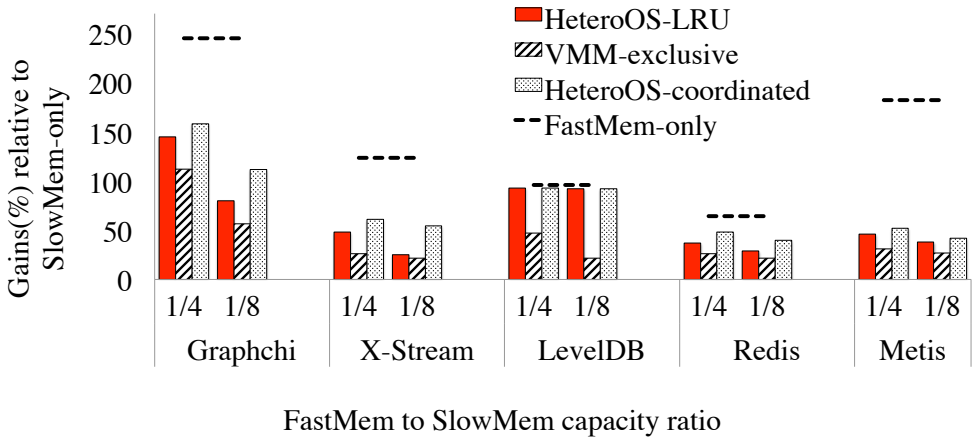


Figure 29: Impact of coordinated management.

memory-aware guest-OS to extract page use and incorporate different on-demand allocation methods to support diverse cloud workloads. The benefits from HeteroOS-LRU also highlight the need for novel page replacement methods in future heterogeneous memory systems.

4.4.3 Impact of coordinated management

We next evaluate the impact of HeteroOS's VMM-guest coordinated management, using the same applications as before, for 1/4 and 1/8 capacity configurations. In Figure 29 the y-axis shows the speedup relative to SlowMem-only, and the dotted lines represents the best case FastMem-only approach. Figure 11 compares the gains only from migrations and the total pages migrated (in millions) for three applications (for brevity) relative to to the Heap-IO-OD which completely relies on smart page placement without any migration. In Figure 29, we compare the following approaches described in Table 10: (1) VMM-exclusive with hot page scan of 16K guest-VM pages in a 100msec interval, (2) HeteroOS-LRU - the best case approach analyzed in Figure 27, and (3) the coordinated approach where the guest guides the VMM to scan only relevant pages from the applications' address space instead of entire guest-VM, and to dynamically vary the hotness scanning interval from 50ms to 1 sec based on change in cache misses using Equation 1 in Section 4.3.

VMM-exclusive vs. HeteroOS-LRU: The VMM-exclusive approach completely relies on page migration and performs poorly for most applications for the following reasons. First, FastMem pages are not allocated promptly, even when free FastMem pages are available.

Table 11: Migration gains relative to Heap-IO-OD. Values in braces indicate pages migrated in millions.

Apps.	VMM-exclusive	HeteroOS-LRU	Coordinated
Graphchi	-30.0 (0.69)	10.0 (0.10)	40.0 (0.33)
Redis	-20.0 (0.51)	2.1 (0.11)	19.0 (0.26)
LevelDB	-10.0 (0.14)	20.0 (0.01)	20.0 (0.08)

Second, uninformed migration of short-lived I/O pages adds unnecessary but significant migration cost for applications such as X-Stream and Graphchi. Third, the VMM lacks the information to evict FastMem pages that are released inside the guest-OS resulting in ∼40-45% lower page reuse compared to a HeteroOS-LRU. In contrast, HeteroOS-LRU uses on-demand allocation when possible and resorts to LRU-based eviction and migration only when free pages are unavailable. Finally, applications with small WSS (e.g., I/O-intensive LevelDB) do not benefit from hot page detection and migration. For Metis with large WSS and heap-only access, the VMM-exclusive results are comparable to HeteroOS-LRU results.

HeteroOS-LRU vs. HeteroOS-coordinated: The coordinated approach combines HeteroOS-LRU and VMM's hotness tracking. It achieves benefits by using guest-OS hints and hardware counters to limit the scope and frequency of hot-page tracking and migration. As shown in Figure 29, the coordinated approach outperforms the HeteroOS-LRU (only the guest-OS involved in management) for memory capacity-intensive and cache-intensive applications constrained by FastMem size. For instance, HeteroOS-coordinated provides 158%, and 61% improvement gains for Graphchi and X-Stream, respectively, even with a 1/4 capacity ratio. In contrast, for LevelDB, with a WSS that fits into the available FastMem, there are no gains from the coordinated approach. In addition, as shown in Table 11, the coordinated approach increases the number of page migration only for applications constrained by FastMem capacity compared to HeteroOS-LRU evictions, but more importantly, the performance gains only from page migrations are significant.

4.4.4 Weighted DRF-based resource sharing

We compare a DRF-based HeteroOS-coordinated, which provides max-min fairness for more than one memory type, with the VMM-exclusive and HeteroOS-coordinated approach using the default single resource max-min fairness approach.

We run Graphchi and Metis in separate VMs on a system with 4GB FastMem and 8GB SlowMem, divided between two VMs. Graphchi requires a total heap of 6GB memory for execution (see Figure 4), but given its active WSS of 1.5GB, a 1/4 FastMem capacity ratio provides a good speedup. Hence, we use (FastMem: 1GB, SlowMem: 4GB) with resource allocation vector of (2*1GB, 1*4GB), where the values 2 and 1 represent the FastMem and SlowMem weights, respectively. Metis' WSS is ∼5.4GB and a heap of 8GB. Hence, we use rest of the 3GB FastMem with a vector (2*3GB,1*4GB) configuration. The dominant weighted resources for Graphchi and Metis are SlowMem and FastMem, respectively. Note that the single resource max-min provide fairness either for FastMem or SlowMem (FastMem in this result) and when FastMem is fully used, the SlowMem is allocated on-demand without any guarantees.

The bars in Figure 30 show the runtime when multiple VMs are executing, and the star indicates the performance of a single VM with HeteroOS-coordinated, which performed the best in the earlier evaluation. As expected, multi-VM execution increases resource contention and is slower than the single VM case.

Max-min VMM-exclusive vs. HeteroOS-coordinated. First, as shown in Figure 30, even with simple max-min, HeteroOS-coordinated outperforms the VMM-exclusive approach. HeteroOS-coordinated first uses the initial FastMem allocation for each VM, and only afterwards falls on guest-VMM coordination to identify, and migrate select pages to SlowMem, thus increasing the FastMem reuse. The VMM-exclusive approach relies on migrations only, and leads to lower performance. Note that the slowdown for multi-VM Graphchi is significant compared to Metis. This is because Metis is first to exhaust its FastMem limit and then starts consuming a significant amount of SlowMem. This is because a single resource max-min provides fairness for only one resource (FastMem). As a result, Graphchi, which relies on SlowMem beyond its 1GB FastMem, is significantly slowed down.

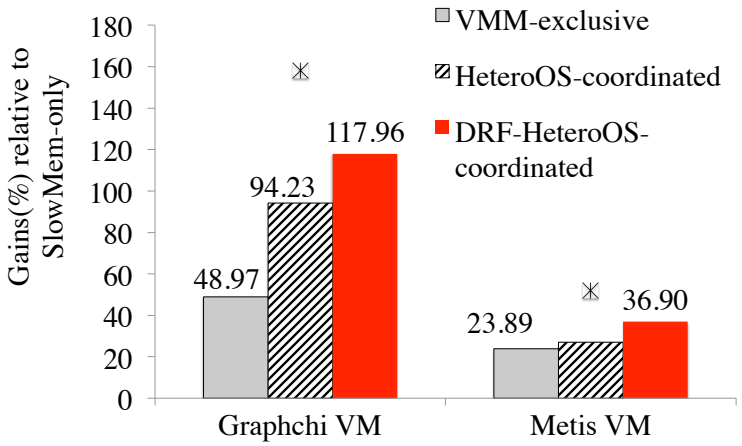


Figure 30: HeteroOS DRF-based multi-VM resource sharing.

Max-min vs. weighted DRF HeteroOS-coordinated. DRF uses a dominant share of each application/VM to provide a multi-resource guarantee. For Graphchi, SlowMem is the dominant share, and it receives the 4GB SlowMem. As a result, DRF avoids significant slowdown of Graphchi and provides 23% and 69% improvement over max-min HeteroOS-coordinated and VMM-exclusive, thereby, improves the overall system performance.

4.4.5 Evaluation summary

First, HeteroOS, by making guest-VM heterogeneous memory-aware, extracts page use information for heap, IO page cache, network, and slab buffer to FastMem, thereby, reduces migration, and provides around 180% speedup compared to using only SlowMem. Second, the novel HeteroOS-LRU reduces FastMem contention and increases the gains by 194% (*sim*3x) gains for 1/2 FastMem to SlowMem capacity ratio and 143% gains for 1/4 ratio. Third, HeteroOS-coordinated uses OS-level information to guide VMM's hotness tracking and provides 2.5x gains over SlowMem only approach even with 1/4 capacity ratio, and 2x improvement over to VMM-exclusive approach. Finally, the DRF-based resource sharing provides up to ∼69% gains compared to the VMM-exclusive approach.

4.5 Chapter summary

In this chapter, we explore the use of heterogeneous memory in virtualized systems, for different datacenter applications. We discuss HeteroOS, an OS design that addresses the

drawbacks of existing state-of-the-art VMM-exclusive management solutions. HeteroOS exposes memory heterogeneity to the guest OS, and extracts the OS-level memory usage information to transparently place applications' memory across the heterogeneous memory system, and to improve the overall memory management efficiency. HeteroOS also provides coordinated management among the guest OSes and the VMM, and uses DRF-based algorithm to effectively share resources across VMs. HeteroOS results show up to 3x improvement compared to using only the SlowMem and up to 2x benefits over a VMM-exclusive approach.

CHAPTER V

CACHE MANAGEMENT FOR HETEROGENEOUS MEMORY

The previous chapters focused on the OS and the hypervisor support for managing heterogeneous memory by extending the virtual memory. As discussed in Chapter 3, future byte addressable non-volatile memory technologies (NVM) such as phase change memory can be used for both increased memory capacity and for fast access to persistent storage termed as dual use in this chapter. Specifically, for end-user devices such as smartphones and tablets, dual use of large capacity heterogeneous memory such as NVM can satisfy both capacity and persistence benefits. In fact, dual use can reduce the need for larger SSDs. However, dual use of NVMs results in cache interference between persistent and capacity use. We discuss methods to overcome this issue.

5.1 Introduction

In this chapter, we address the challenge of how to efficiently use NVM's byte addressability, in terms of bypassing software stack overheads, while at the same time, enabling persistence for such memory when and if desired. Referring to the use of NVM for additional heap capacity without persistence as *NVM capacity* vs. its use of persistence as *NVM persistence*, we contribute (1) detailed studies of the performance overheads of simultaneously exploiting these two capabilities of NVM, followed by (2) the creation and evaluation of techniques that mitigate these performance costs. Specifically, concerning (1), NVM's high write latency compared to DRAM (5x-10x) [18] makes it difficult to use it for extended capacity – NVM capacity. Obtaining Comparably high performance requires the efficient use of system caches by the end client (end-user) applications being run. But the cache is also used when using NVM for persistence where to guarantee consistency, durability, and failure recovery, the application data as well as its metadata must frequently be serialized and flushed from the cache. Cache line flushes involve write-back of dirty data (if any) and cache lines invalidation broadcasts across all cores. Further, since evictions from the cache

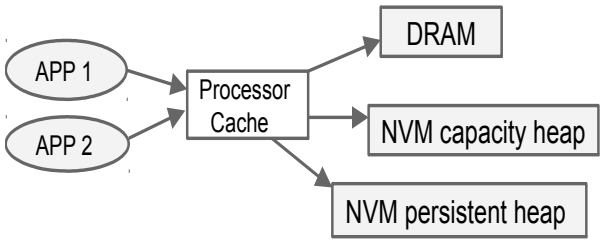


Figure 31: Dual-use NVM high level model.

can be in any order, the updates from cache must be serialized, by fencing memory write operations [146, 57]. An expected outcome of such actions is increased cache misses and higher NVM access latency for NVM persistence-based applications. A perhaps less obvious, yet quite undesirable outcome is that such cache flushes can also substantially impact the NVM capacity applications using the same last level cache. In this chapter,

1. Persistence Impact: we analyze end client device workloads to understand better the impact of NVM persistence applications on NVM capacity applications sharing the same cache.

2. OS-level Cache Sharing: to reduce cache misses due to sharing, we propose a novel but simple page coloring mechanism that exploits as a metric 'physical page contiguity misses.' The approach is implemented in the Linux kernel memory management layer and reduces cache misses by 4% on an average, validated through hardware performance counters.

3. Library-level Optimizations: the metadata structures needed for persistence cause overheads in terms of increased cache misses. We analyze persistent memory allocators and their durability-related data structures across the system stack and propose a novel cache-efficient allocator and an efficient hybrid (word-object) logging approach that significantly reduces the number of writes to NVM.

All solutions are evaluated with standard benchmarks and with the realistic end user device workloads.

5.2 Background and related work

Role of cache. For NVM-based persistence, high write latency overhead can be mitigated by using a fast intermediate cache and DRAM, as shown in [125, 88, 60]. For a write-back cache, writes that are evicted from the cache are then moved to the DRAM cache that behaves like a disk buffer cache. This model works well when there is sufficient buffer space (e.g., on high-end servers), but when DRAM is scarce (e.g., on mobile devices), forcefully reserving pages for buffering can reduce overall system throughput. The Android OS, for instance, avoids DRAM use for page buffering by disabling swapping. A better alternative is to use the processor cache [146, 56]. The unfortunate consequence, however, is that consistency and durability guarantees require it to be frequently flushed from the cache. We adopt this approach, contributing a thorough study of its performance implications.

Software support for NVM. The usage model for NVM – capacity extension, persistent storage, or dual-use – determines the systems software support needed for NVM management. (1) *Application involvement.* For NVM capacity, the system can treat NVM as a swap device [60, 98, 88], not involving applications. However, this is not the case for NVM persistence, which requires applications to identify specific data structures explicitly. A recent work [73] proposes using NVM by intercepting the memory access to specific ranges, and a modified NVM controller to redirect access to NVM. The key contribution of this work is to provide atomicity and durability guarantees with efficient use of a cache.

Applications use of NVM capacity and NVM persistence. The number of cores and application threads in end-clients is increasing along with the increasing DRAM capacity and storage requirement. For instance, take the case of a multithreaded memory hungry web browser where the front end browser tab uses NVM capacity for an additional memory buffer, and the backend browser thread caches user data to the transactional database. Similarly, in a multi-threaded game engine, the GUI thread can use NVM capacity as graphics buffer, and the game I/O thread can access NVM persistence for storage (load/store user state to the database). For NVM capacity-based allocation, the OS NVM manager does not track application (user-level allocator) and kernel data structures, but simply allocates pages (like DRAM). But for NVM persistence, both user and kernel data structures are

tracked. Hence, for NVM persistence, applications use explicit NVM allocation interface, whereas the capacity use of NVM is transparent (by linking to NVM library) without requiring any application level changes.

Durability and consistency. NVM hardware-software must support required consistency and durability across application sessions. When using the processor cache to hide the high write latency, the cache data can be evicted in any order to maintain ACID guarantees. Prior efforts use write-through caches [146], or epoch-based cache eviction methods [57] using memory barriers for ordering NVM writes. Further, durability can be affected by power failures or device crashes, leaving the application in a non-deterministic state, e.g., due to partial updates (note that both data and metadata must be saved consistently). A standard approach to deal with this relies on application commits, which in turn trigger cache flushes. Frequent flushes can reduce the possibility of non-recoverable failures, but additional transactional mechanisms are needed for atomicity, accompanied with logging (e.g., undo/redo) support for durability. A recent work [160] proposed hardware-based nonvolatile cache and nonvolatile memory to enable multi-version support with in-place updates (avoids the logging cost). The cache contains the dirty version and the memory contains the cleaner version. While such micro-architectural changes can reduce the cost of logging, we focus on the software optimizations for existing hardware (volatile cache).

5.3 The costs of persistence

This section motivates the need for cache efficient end-to-end solutions when using NVM both for extended capacity and for persistent storage. We analyze the impact of cache sharing between persistent and non-persistent applications. Our analysis shows that supporting ACID-based persistence is a primary factor contributing to higher cache misses. These inefficiencies arise in both the memory allocator and the logging mechanisms. To analyze the overheads, we implemented a complete NVM software stack (application library). In keeping with the end client focus of this research, all experimental evaluations use a dual core 1.66 GHz 64 bit D510 Atom-based development kit running a 2.6.39 Linux kernel with our OS-based NVM support, with 2GB DDR2 DRAM, Intel 520 120GB SSD, 32KB L1

and eight ways 1MB L2 write-back cache [6]. Out of the total 2GB system RAM, 1GB is used towards NVM. MSR-based performance counters are used to measure cache misses, and the VTune analyzer is employed for function level miss estimation, for both user and kernel code. Applications are run with almost same execution times to capture the effects of resource sharing.

5.3.1 Impact of unmanaged cache sharing

Representative end client workloads are used to verify the performance penalties of sharing last level cache across NVM capacity and NVM persistence applications, the former using NVM for additional capacity but co-running with a single additional NVM persistence workload – a persistent hashtable (e.g., like those used in key-value stores), labeled as PHT (Persistent Hash Table). Again, the NVM capacity-based applications do not require data persistence and thus, do not flush state of the cache, whereas the PHT NVM persistence-type application frequently flushes cache to obtain consistency and durability guarantees. Figure 32 shows the pseudocode of the PHT with strict transactional guarantees, which we implement using a transactional heap library that ensures consistency and durability for applications. The pseudocode lines marked in red indicate the need for cache flush. The X axis in Figure 33a shows representative end client NVM capacity applications. X264 is a video conversion application converting a 50MB '. avi' to a mobile compatible '. mp4' file. *B-tree* is a cache efficient data structure commonly used for clients' in-memory databases. Animate provides animation for image files. Each such NVM capacity application is co-run with the PHT with random puts and gets for 500K keys (this size is based on our experimental device's available memory resources). The Y axis shows the cache miss (%) increase of co-running these NVM capacity applications with the NVM persistence hash table relative to running NVM capacity application with a version of the hashtable that is not persistent – NVM capacity. Results obtained from reading MSR performance counters demonstrate that while cache efficient NVM capacity workloads like B-tree are not heavily impacted by the presence of the NVM persistence application (PHT), codes like X264 and animate suffer a substantial increase in cache miss rates. Also of interest is the high

variability of cache miss rates for NVM capacity, an unintended side effect of co-running NVM capacity with NVM persistence applications.

We also validate our previous analysis using a cycle accurate instruction level architectural MACSim simulator [13]. We use CPU intensive workloads for this study and replace the I/O intensive animate use case with the end user benchmark WebShootbench [28], a well-known workload used by Google for Chrome OS tablet benchmarking. Also used are some memory intensive and CPU intensive SPEC workloads, since our goal is to investigate the impact of dual-use of NVM. For modeling the cache impact, we modify the simulator to identify all cache flush instructions in the trace generated by the PIN tool, invalidate those cache lines and write-back the cache lines if they are dirty. We use write-back cache as prior work [111, 146] have evaluated the performance impact due to write-through cache. As seen in Figure 33b, most of the memory intensive benchmarks show substantially increased cache misses and write-backs when co-running with the PHT. Simulation results report only the cache misses incurred by applications, whereas the hardware counter-based measurements using the Intel VTune analyzer in Figure 33a also report cache misses due to OS functions (kernel mode execution of the application), constituting about 11-16% of the overall cache misses observed. The clear conclusion from these experimental evaluations is the need for effective ways to reduce the impact of NVM persistence applications on co-running NVM capacity applications, particularly given the ever-increasing number of concurrent applications being run on today's end user devices. One way forward is described in Section 5.4.

5.3.2 Library overheads

Preventing NVM persistence applications from impacting the performance of NVM capacity applications requires end-to-end solution that begins at user-level, for two important components: (1) The memory allocator used by all NVM applications and (2) the logging manager guaranteeing durability for NVM persistence codes. They are each discussed below.

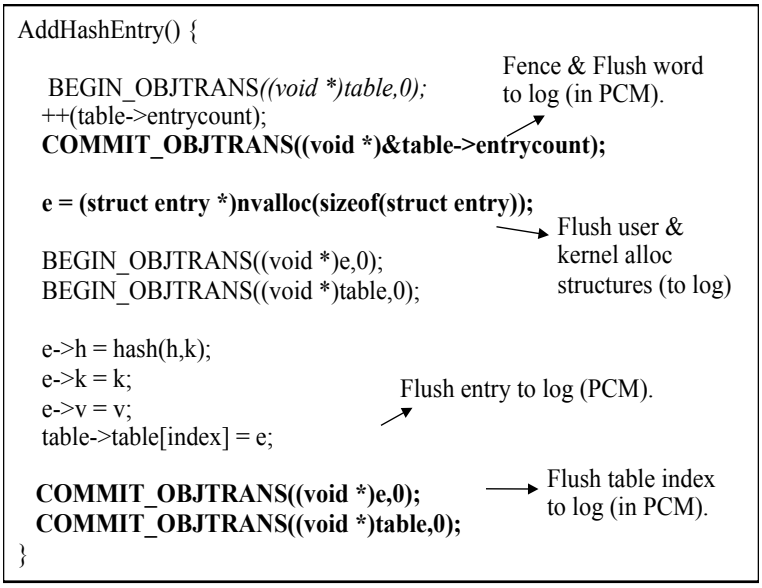
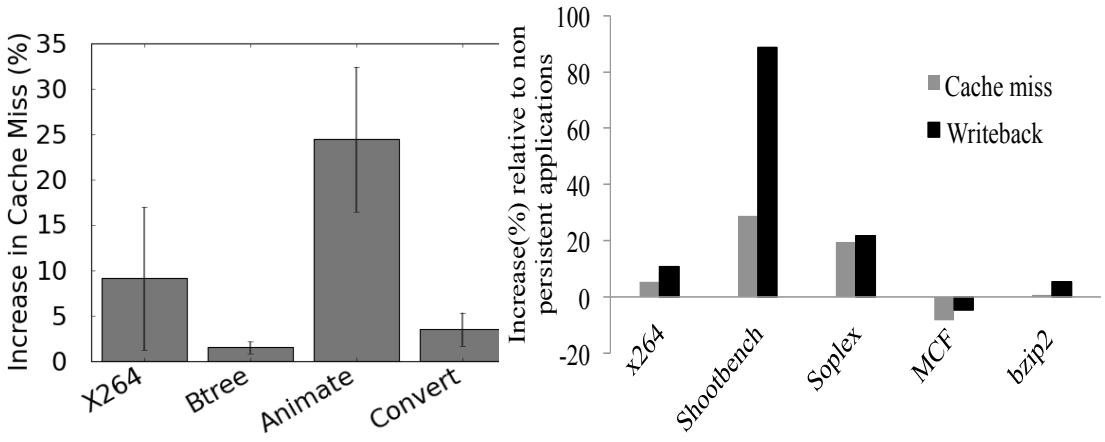


Figure 32: Transactional persistent hash table Insert.

5.3.2.1 Cache inefficient persistent memory allocators

The allocator strongly influences application performance, particularly for data structures requiring frequent allocations (e.g., tree structures, linked lists, key-value stores using hash tables, etc.). Modern allocators, however, maintain complex hierarchical metadata structures for fast, free space lookup, object (malloc'd memory) deletion, and more importantly, for reducing fragmentation. Jemalloc (see Figure. 34), for instance, is a multithreaded cache efficient allocator that allocates large regions of memory, called chunks (1024 pages per chunk), where each chunk is further divided into 'pageruns.' Each pagerun maintains a class of uniformly sized objects that vary from 8 bytes to 512 KB. Every page run has a fixed number of equally sized objects. The page run has one header with a bitmap to indicate used and freed objects. When an application allocates memory, based on the requested size, a corresponding pagerun is selected, and checks for free objects and the corresponding bitmap and page run header are updated. For a group of objects in a pagerun, one header and a bitmap are sufficient. The allocator data structure and application data are placed separately, to keep the application data contiguous and reduce cache misses on application data. For efficient memory usage and to reduce fragmentation, the allocator's metadata is frequently updated.



(a) Full run. **(b) Simulator Analysis.**

Figure 33: Impact of co-running NVM capacity with NVM persistence hashtable (1.5M operations).

Most prior proposals, to the best of our knowledge, maintain all allocator metadata in NVM [111, 56, 146]. However, keeping such frequently updated data on NVM results in a significant number of writes to NVM [111] thereby impacting performance. Further, compared to volatile object allocations, additional metadata is required for each persistent object. For non-persistent objects, the current virtual address is sufficient to locate an object in a pagerun and update its metadata. But for persistent objects, the virtual address is invalid across restarts. Hence, objects contain additional information to locate them and identify their commit status (some prior work [111] even maintains CRC with each object). Furthermore, every update to the allocator data must be logged and flushed from the cache. Such cache flushes write dirty lines if they inconsistent with memory broadcast the invalidation across cores. These steps can generate high cache misses, resulting in direct writes to the NVM and application slowdown. In summary, frequent allocator metadata updates will result in substantial 'persistence cost' (e.g., consider a PHT with millions of new entry addition and deletion). Section 5.4.2 shows solutions that improve upon metadata structures and updates to mitigate these problems.

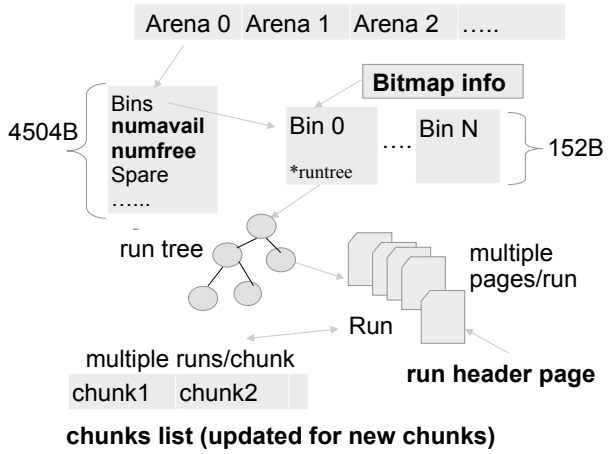


Figure 34: Jemalloc data structure. Rectangular blocks represent C structures. Fields in red are frequently flushed when metadata is in NVM.

5.3.2.2 Durability-based write latencies

For providing ACID guarantees to applications, the NVM stack must support transactional semantics, coupled with a fail-safe mechanism where every change to application's memory must be logged. Logging mechanisms are used for recovering from failure to a consistent state. They are broadly classified into 1.) UNDO and REDO methods, 2.) and based on the logging granularity, as a word- vs. object-based. For UNDO logging, before every write to the log, the stable version is first copied to a log, after that the application can continue writing to the original data location. If a transaction fails, recovery actions copy the stable data from the log back to the original memory location. For REDO logging, all writes are appended to a log, and when a log fills up, the log entries are copied to original memory locations. Regarding the logging granularity, prior NVM works uses either (i) word-based logging [54, 133, 146] where each word is logged along with log metadata (described shortly) or (ii) an object-based log for NVMs [56], where the entire object is copied to log. We next discuss problems with this current state of the art.

Each log entry is a record consisting of a metadata and the actual data stored in different locations. The record contains the actual word address, a pointer to the data in a log, and a pointer to next log record. To log a word of data (8 bytes), 24 bytes of data must be

written to NVM, thereby drastically increasing the overall writes to NVM. Further, word-based logging requires substantial rollback time (scan word by word and apply updates). Recent work avoids repeated updates by logging at object granularity [33, 56]. While this scales well for large objects, updates for smaller member variables or counters of an object (e.g., updating a counter in a hash table structure when new entries are added), the object copy cost from its actual address to the log (in case of UNDO) or back from a modified object to an actual data address (in case of REDO) can be substantial. Further, cache misses increase with increasing object copy sizes, resulting in slower NVM access. Section 5.4.3 describes a novel hybrid logging approach that combines word-based and object-based logging to provide an adaptive approach for reducing NVM write latency issues. The proposed approach does not require substantial developer effort to classify word- and object-based logging.

5.4 NVM-efficient end-to-end software solutions

This section describes solutions to the cache inefficiencies identified in the previous section. It first describes a physical page contiguity-based page allocation mechanism that seeks to partition the cache entries used by NVM persistence vs. NVM capacity applications. This simple but effective solution avoids the complexity of using traditional page coloring methods for this purpose. Second, allocator metadata management is improved to reduce allocator overheads, the key idea being to maintain complex allocator metadata structures in DRAM and logging their updates in NVM. Third, we reduce the cost of transaction logging via a hybrid logging mechanism that automatically adapts to the appropriate logging granularity (word vs. object). While our cache partitioning mechanism reduces the cache impact of NVM persistence apps on NVM capacity apps, the allocator and logging optimizations reduce the cache misses suffered by NVM persistence applications and hence improves the overall cache misses. Experimental evaluations of the effects of each such NVM write-aware optimization are run on the same Atom-based development platform described in Section 5.3, using the same methods to gather experimental results via the MSR performance counters. All evaluations use realistic end-user NVM capacity applications that

include 'animate' used for image animation, 'x264' used for 64 MB file mp4 conversion, 'convert' used for converting images from jpeg to bitmap (png) format. We also use SPEC 2006 benchmark applications that are more representative client-side applications, like 'astar' which is a portable 2D path-finding library, 'povray', a ray tracing application, and other memory intensive benchmarks like 'omnetpp', 'mcf', 'soplex' and computation intensive benchmarks like 'sjeng', and 'libquantum'. Details about these benchmarks concerning their cache and memory intensity can be found in [90].

5.4.1 Reducing the impact of cache sharing

Cache sharing between NVM capacity and NVM persistence applications can result in increased conflicts, false sharing, higher NVM writes, and reduction in memory bandwidth due to increased NVM traffic. We hypothesize that increased cache misses experienced by NVM capacity applications, caused by co-running NVM persistence application, can be avoided by partitioning the shared cache across these applications. Hardware and software cache partitioning strategies for multiprocessor systems have been extensively studied in the past. Hardware mechanisms [112, 51, 124, 143] include simple static as well as dynamic partitioning methods, the latter monitoring the cache miss suffered by applications, and then adjusting the number of cache ways between NVM capacity and NVM persistence applications. Software-based partitioning approaches [159] typically use page coloring mechanisms, which we describe shortly. Generally, hardware approaches have shown higher benefits [158] compared to software partitioning, but software partitioning provides the flexibility to enable easily or disable page coloring and offers scope for various application-specific optimizations.

We study the effectiveness of cache partitioning using the cycle accurate MACSim simulator, by statically partitioning one 1MB LLC cache to use 3/4 of the cache sets dedicated to NVM capacity application, and 1/4 for the NVM persistence applications. The analysis uses 500 million instructions of the same applications as those used in Figure 33b. Note that most memory-intensive applications show up to 12% improvement in performance, gained from cache partitioning, while there is no or little impact on other benchmarks like bzip,

and MCF.

Page coloring. OS-based cache partitioning between applications using software page coloring has been studied extensively. Implementing page coloring perfectly requires substantial changes to the memory management layer of the OS [61]. Further, even when such changes are present, due to increasing cache associativity, operating systems often disable the page coloring feature. For end client devices with their few way caches (4-8 ways) and given our diagnosis of high conflict misses with NVM capacity and NVM persistence applications, however, we posit the need to revisit page coloring. We, therefore, prioritize two goals for suitable OS-based cache partitioning methods: (1) to reduce the complexity of page allocation (i.e., to avoid looking for specific pages at the time of page allocation) of existing, low overhead page coloring mechanisms [61, 158], and (2) to make it easy to disable OS-based partitioning when only NVM capacity applications are currently running (i.e., no co-running persistent applications), or when there is little or no impact of persistent on non-persistent applications.

Page contiguity-based partitioning. We propose a novel adaptive method for cache partitioning that leverages the high probability with which current caches map contiguous physical pages to contiguous cache lines. The key idea is to increase the physical contiguity of pages allocated to an application. Intuitively, the more contiguous an application page, the more contiguous its cache lines, and the less likely the cache interference with other applications sharing the same cache. This is in contrast to non-contiguous allocations in which different applications' random pages are mapped to various cache lines, thus increasing the chances of cache conflicts.

Allocating a single page during first touch and page fault can result in high page contiguity misses. We call this policy JIT (just in time) allocation. For example, let N be the number of applications simultaneously accessing/allocating NVM pages. With JIT allocation, the probability of an application receiving a physically contiguous page reduces to (1/N). Hence, with an increasing number of co-running applications, cache conflicts increase. Reducing the number of a physical page (physical frame) contiguity miss can substantially reduce cache misses. To increase the contiguous pages allocated to the application is that,

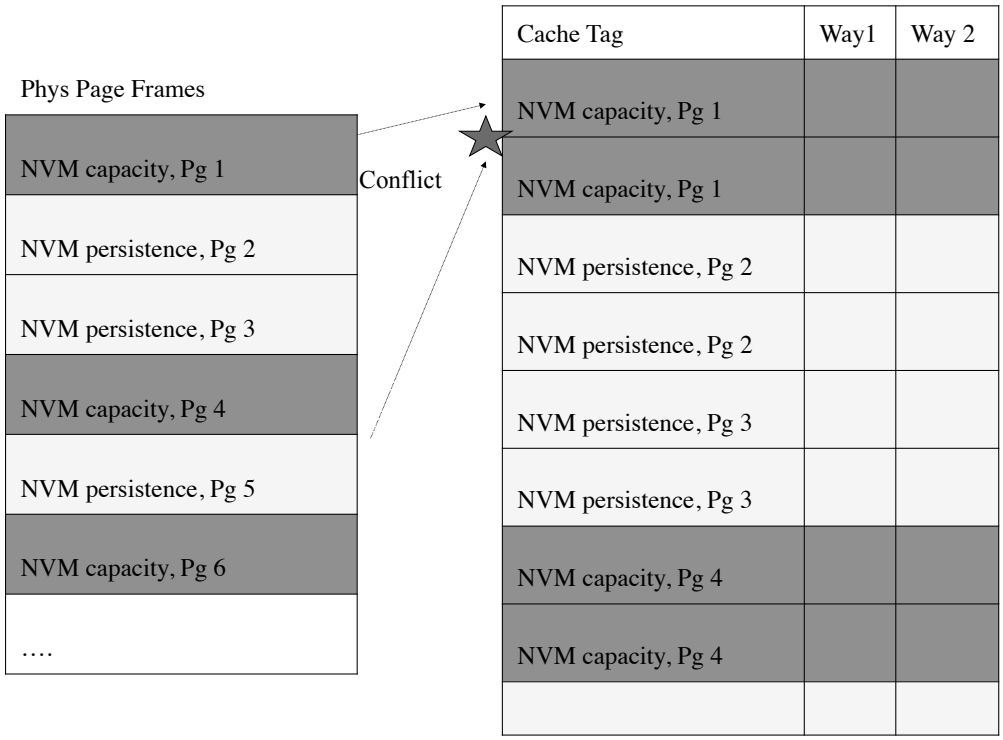


Figure 35: Cache conflicts due to JIT allocation. NVM capacity page and NVM persistence page maps to same set. In this example each physical page maps to 2 cache sets.

during page faults (minor faults on the first touch), instead of allocating one page, a batch of physically contiguous pages is reserved for the application, and on subsequent page faults, specific physical contiguous pages are added to the page table. We refer to this approach as contiguity-aware allocation (CAA).

Implementation of CAA. As a first step, to avoid contiguity misses, we create two types of lists in the kernel: 1) a contiguous list, and 2) a non-contiguous list. Each list contains one or more buckets. Each bucket in the contiguous list contains an array of physically contiguous pages. For instance, Figure 37 shows a contiguous list with three buckets and each bucket contains four physically contiguous pages. A list here refers to linked list of buckets. Buckets in the non-contiguous list (the list at the bottom of the figure), contain pages that are ordered but not contiguous.

For adding contiguous pages to the bucket, when there is a page fault, a batch of contiguous physical pages is allocated and added to an application specific bucket in the

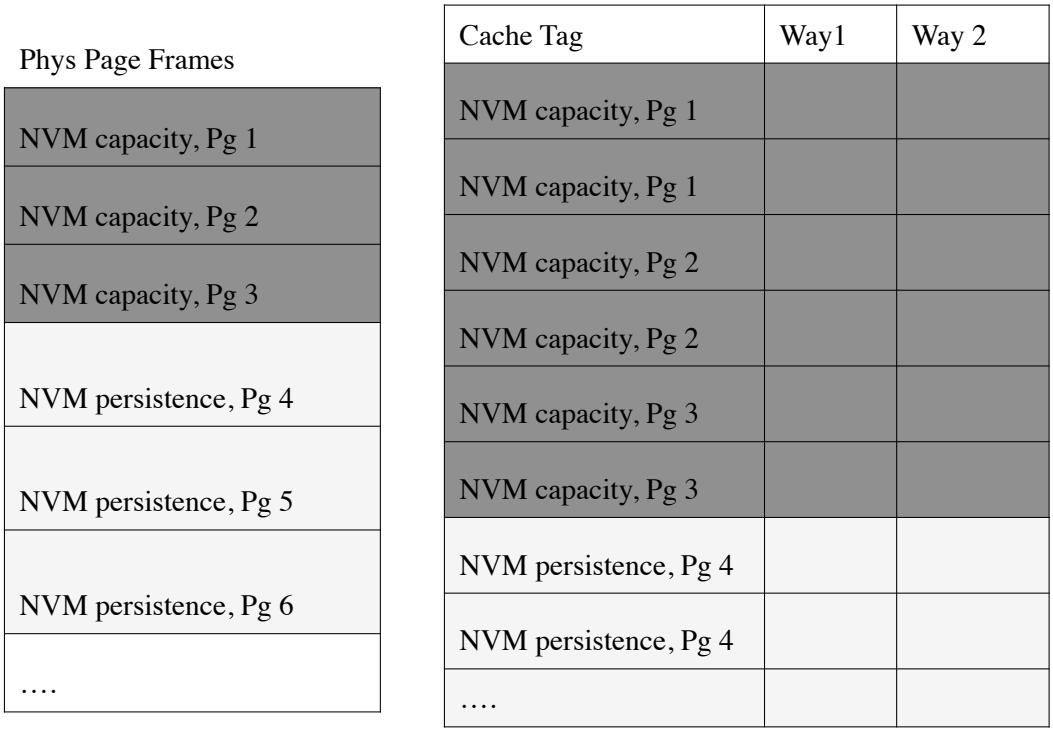


Figure 36: Reducing conflicts with contiguous page allocation.

contiguous bucket list. While only the page corresponding to faulting address is added to page table, the other contiguous pages are used during subsequent page faults thereby using physically contiguous pages as shown in the Figure 36. Our design creates a separate bucket for each application, and as the pages of the buckets are exhausted, new batch allocations refill the bucket. It is not always possible that contiguous batch allocations succeed (and depends on memory availability). If batch allocations return non-contiguous pages, such pages are moved to a bucket in the non-contiguous list. We avoid creating multiple buckets so as to increase the locality of the bucket data structure in the cache, but can easily support multiple buckets per application thread.

Our approach divides the applications into cache friendly and non-cache friendly applications. All applications are allocated from contiguous buckets initially. We maintain two memory watermarks (higher: less critical, lower: highly critical). As the free available memory reaches less than the 'higher water mark', we start using the non-contiguous pages for NVM persistence applications, and when the memory limit reaches less than the

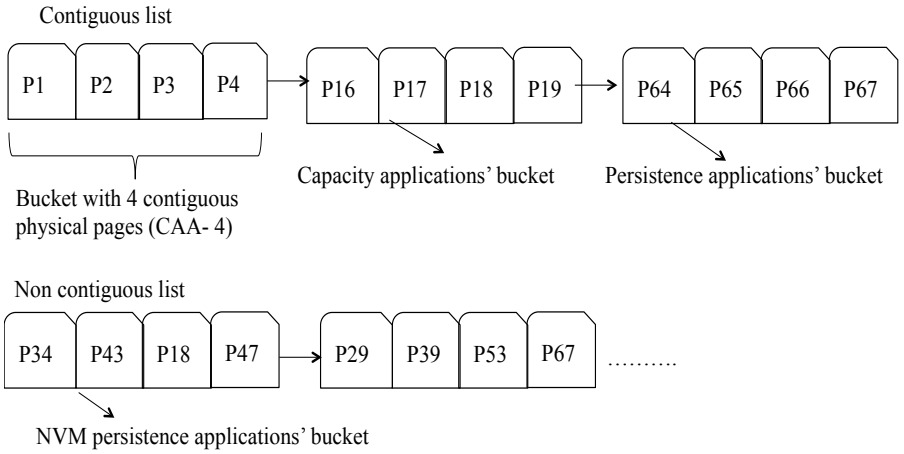


Figure 37: Page contiguity bucket-based design. PX in the figure denotes physical page number X.

low water mark, we disable page contiguity-aware allocation. Our approach significantly reduces the page contiguity misses and also the cache misses due to conflicts as shown in our evaluation.

Evaluation baseline. In all evaluations, as a baseline, We use the PHT (persistent hash table) as the NVM persistence application. The PHT uses the current JIT-based OS page allocation combined with a naive allocator that stores/access all its complex allocator metadata in NVM, and uses a word-based logging as proposed by prior 'NVM as a heap' research [146]. We also report average (across all workloads) cache miss reduction after applying each optimization.

Page contiguity miss analysis. We next evaluate the effect of bucket size (the number of contiguous pages) on the page contiguity misses. In Figure 38, the x-axis shows several client and SPEC benchmarks, and the y-axis shows the percentage of reduction in physical page contiguity miss compared to JIT allocation. We evaluate our experiments for two different bucket sizes – CAA-4, CAA-16, where 4 and 16 indicate the number of contiguous pages allocated in a batch and added to a bucket in the contiguous list when handling a page fault. As seen from the results, we can reduce the physical contiguous page misses for applications by up to 75% for CAA-4, and 91% for CAA-16, for both client and SPEC workloads, compared to the JIT-based method. Table 12 shows the actual number of page contiguity misses for some of the applications using all the three (JIT, CAA-4, CAA-16)

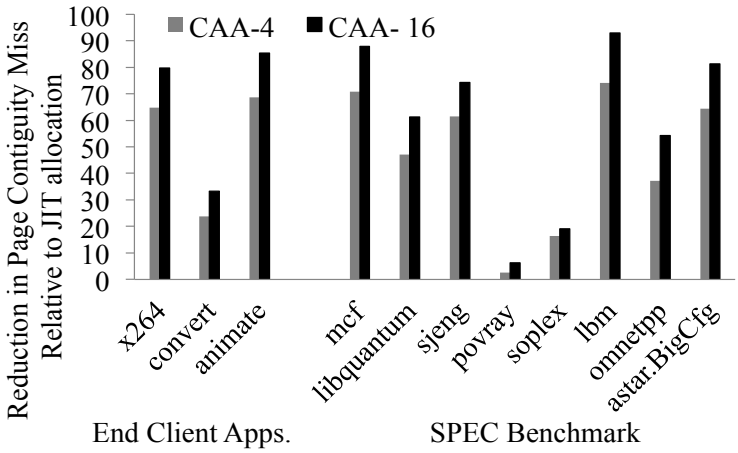


Figure 38: Page contiguity miss analysis, CAA-16 and CAA-4 indicates CAA with 4, 16 pages/bucket.

Table 12: Page contiguity miss count.

App	JIT	CAA-4	CAA- 16	Expected benefits
x264	4890	1720	990	Low
convert	177340	135080	118360	High
animate	4860	1520	710	Low
povray	8160	7950	7640	Low
soplex	989560	828080	801090	High
astar.BigCfg	107360	38310	20120	High

methods. Applications like x264, animate, sjeng, show a higher reduction in miss counts, however, their expected benefits due to physical contiguity can be small since the total contiguity miss for these applications is substantially lower even when using JIT-based allocation. In comparison, for memory-hungry applications such as convert, soplex, astar, the benefits due to page contiguity is higher. We next evaluate whether page contiguity miss reductions results in reduced cache misses of NVM capacity applications.

Reduction in overall cache misses. Figure 39 shows the relative reduction (in percent) in cache misses for the same set of benchmarks compared to the baseline. We use two different contiguity-aware allocation bucket sizes, CAA-4, and CAA-16. We observe that with CAA-4, the benefits in cache miss reduction vary from 1% to 8%. While improvements are observed for most applications, for some applications like x264 and soplex, the misses increase compared to the baseline. Maximum gains are seen for memory-intensive astar,

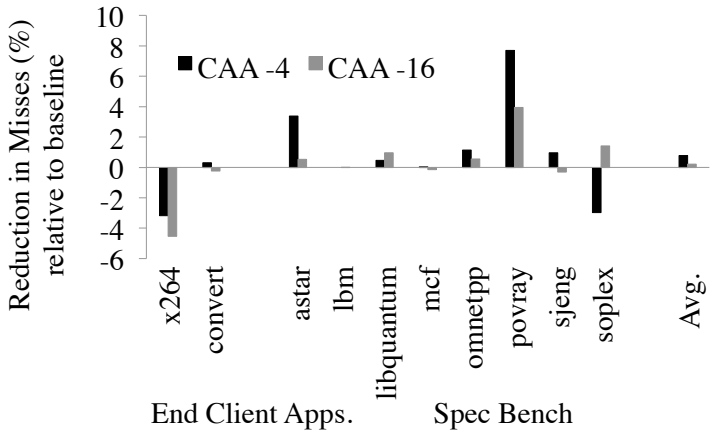


Figure 39: Overall cache miss analysis.

and for the moderately intensive povray. Also, when varying the bucket size from CAA-4 to CAA-16, for applications with smaller memory footprint, such as animate and sjeng, the cache misses increases. Page contiguity for these applications does not provide high gains as the number of pages allocated by these applications is relatively small, and the impact due to cache sharing is relatively less. Also, the working set of the applications fit well into the cache. When using CAA-16, during page fault, the kernel tries to allocate contiguous pages in a batch and then adds them to the bucket linked list. The traversal across linked lists to add new pages or remove free pages incurs cache overheads without benefits from physical page contiguity. Such effects are higher when the bucket size is further increased. For applications with larger memory footprint (soplex, libquantum, omnetpp), CAA-16 based allocation provides more reduction in cache misses compared to CAA-4. These results emphasize the fact that *physical page contiguity-aware allocations reduce the cache sharing impact of NVM capacity and NVM persistence applications and that using the right granularity of bucket sizes (i.e., CAA-4 vs. CAA-16), based on applications' memory usage, can lead to higher cache miss reduction.* The average reduction in cache misses due to CAA is around 1%. We do not run the animate benchmark due to its high memory requirements of the Atom platform. For further improvements, we next focus on NVM write-aware allocators.

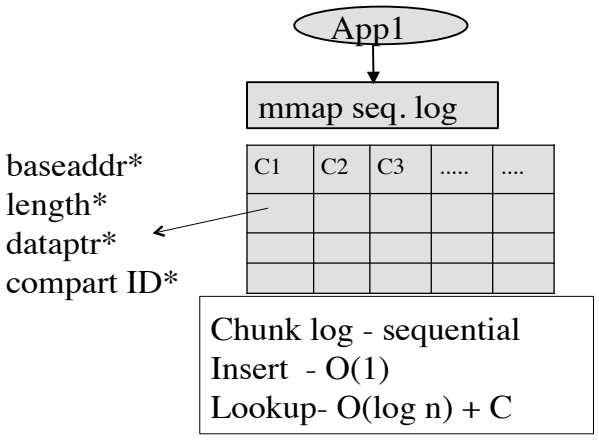


Figure 40: Two level log-based NVM allocator. Avoids complex data structures (Figure 34) in NVM. Fields with star denotes variables flushed to PCM.

5.4.2 Addressing allocator overheads

Cache misses and write-backs due to application allocators can be reduced with an NVM write-aware allocator (NVWA). The key idea is to keep the complex hierarchical allocator metadata in DRAM and maintain only an allocator data structure-independent log in NVM. The log of all NVM allocations, deletions, or re-allocations by application threads is sufficient for restarting and rebuilding the allocator state. The allocator log is always kept strongly consistent with the actual DRAM metadata state by flushing to NVM. The log data structure in NVM is written sequentially, and writes are aligned with cache boundaries.

Figure 40 shows a two-level allocator log. For every memory allocation by an application, the first log level contains information about allocated chunks, and the second level contains information about the physical page to locate the corresponding chunk. For deletion of a memory chunk, there are two possibilities based on available storage space: (1) the log can be parsed sequentially, and the corresponding entry is invalidated (by marking a exclude bit for garbage collection), or (2) one can just 'append' chunk deletion information to the log. For (1), when there is insufficient NVM space, to avoid sequential log parsing, we maintain an in-memory (DRAM) red-black tree with chunk pointers, which makes it possible to locate and update chunk status in $O(logn)$ time. For example, just appending a log entry for a new chunk and for deleting a new chunk (say a million entries hashtable) can consume twice

Table 13: Cache line flush count comparison.

No. of hashtable elements in millions	Naive	NVWA
.5	2500036	500032
1	5000044	1000043
1.5	7500052	1500054

the allocated data size. Further, these updates can cause an additional cache miss overhead. When NVM space is not a constraint, method (2) above can be used. For deletion, only a single bit is modified to indicate whether the chunk is still useful, and for reallocation, only the length field is updated.

The benefits of such a log-based approach are:

- During most metadata updates, no more than two cache line flushes are required, unlike with prior research [111] in which a hierarchy of allocator data is maintained and flushed to NVM.
- Updates to NVM logs are mostly sequential, cache aligned, and hence, no more than two cache line flushes are required.
- Maintaining simple persistent restart metadata independent from the allocator metadata provides the flexibility of using different allocators.
- A final benefit is a reduction in the total amount of persistent data kept in NVM because only the log is persistent.

Allocator overheads. We next analyze the implications of the allocator optimizations using microbenchmarks and representative client applications.

Microbenchmark analysis. We run only the persistent hash table (PHT) benchmark using the default persistent Jemalloc allocator referred as a 'naive allocator' that keeps all of its complex allocator structures in NVM. This is compared to the cache misses seen with the PHT using the NVM write-aware allocator (NVWA). We vary the number of PHT operations that include random hash table puts and gets. Along the X-axis in Figure 41,

we vary the number of elements in a hash from 600K to 1.5Million, each with key and value of 64 bytes. The Y axis shows the increase in cache miss percentage compared to a baseline ideal PHT that does not flush the allocator metadata. As can be observed, in all cases, the NVWA approach outperforms the naive allocator by around 3%. This is mainly because of the reduced number of cache line flushes by NVWA compared to the naive allocator, with only 2 flushes per every newly allocated memory chunk and a single cache line flush for deletion or resize operations. Table 13 compares the total number of allocator-specific cache line flushes of both approaches. Observe that the NVWA allocator reduces flushes as high as 8x compared to the naive allocator, thus substantially reducing allocator-specific misses.

Application benchmarks. Next, we compare the impact of the NVWA allocator on all of the application benchmarks used in Section 5.4.1. Figure 42 compares the cache misses under different allocator designs. For memory intensive client benchmarks like 'animate', we observe close to a 2% improvement when using NVWA, whereas 'convert' shows less than a 1% improvement. Surprisingly, the lesser memory intensive x264 also experience substantial benefits from using NVWA. This is because x264 processes target files in frame size (192 bytes) granularity. Hence, for a 62MB file, the number of allocation-related flushes are substantial for the naive vs. the NVWA allocator. The 'convert' application, in comparison, experiences a relatively smaller number of allocations. Similar trends are observed in memory intensive benchmarks, where the benefits are higher for memory capacity intensive applications like soplex (4%) and libquantum (2%), but other benchmarks show less than a 1% improvement.

An interesting difference between our representative client applications and the SPEC benchmarks is that with realistic client applications, the total number of allocations is larger, and are done throughout the applications' lifetimes, whereas the SPEC benchmarks (including memory intensive benchmarks), have fewer allocations, mostly grouped in their initial execution stages. This abnormal behavior of the SPEC benchmark means that the impact of interference on NVM capacity applications due to persistent allocations is relatively smaller. We conclude that gains from Using an NVWA depends both on the total

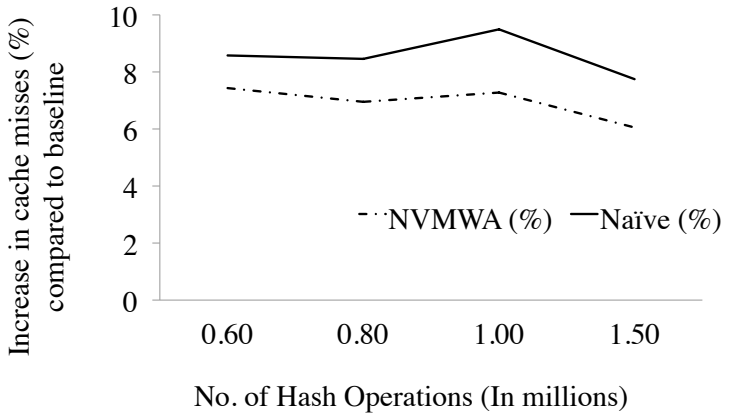


Figure 41: Allocator metadata persistence cost analysis.

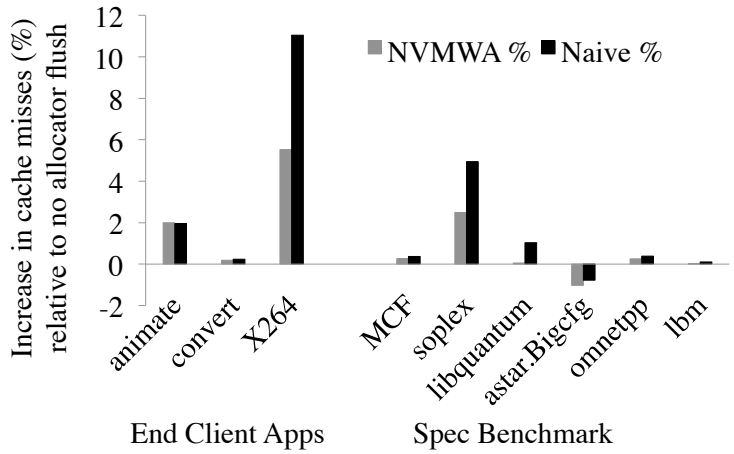


Figure 42: Allocator cache miss reduction (in %) compared to naive approach. JIT is used. (CAA is not enabled).

number of allocations/deletions made by the applications and on when such operations are performed. Figure 43 quantifies the total reduction in cache misses from using the NVWA allocator compared to the ideal baseline.

5.4.3 Hybrid logging

The purpose of the hybrid logging method introduced in this chapter is to reduce NVM writes due to application logging, while still providing with fail-safe durability guarantees. Specifically, hybrid logging (i) reduces the metadata writes for the word-based log (every word data logged requires three words of metadata), and (ii) reduces the data writes of object-based logs in which entire object is copied even when only a single word in an

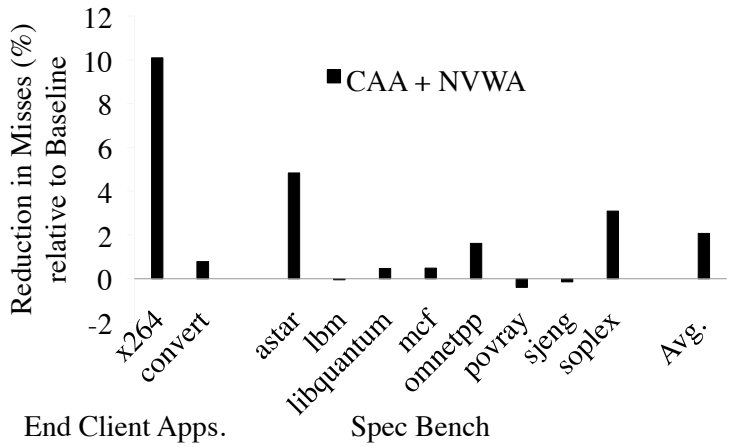


Figure 43: CAA + NVWA performance compared to baseline (JIT + Naive allocator)

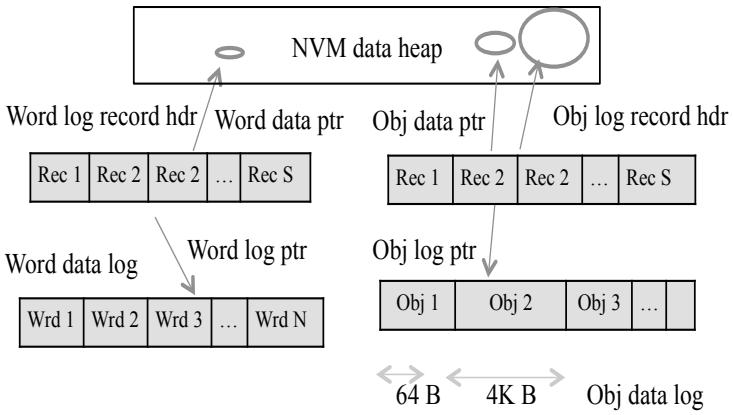


Figure 44: Hybrid logging design.

object has been changed. Hybrid logging provides developers with a flexible object- and word-based logging interface that permits them to pass hints to the write-aware NVM log manager concerning the granularity of changes. In Figure 45, for hash table data structure changes, incrementing the entry count in the hash table or dereferencing to key/value pairs, are word size updates, and the developer can commit these by passing the word logging hint. For changes larger than the word size, e.g., when modifying the key/value object, object-based logging makes it unnecessary to maintain a log record for every word of an object.

Implementation details in Figure 44 show how hybrid logging is used with redo logs (undo

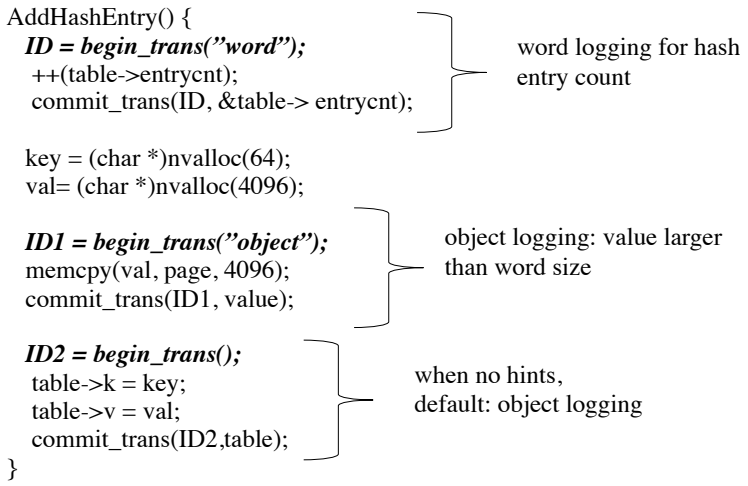


Figure 45: Hybrid logging interface.

logging is supported, as well). The object and word logs for an application are maintained independently. Further, for each log (word and object), the log metadata and actual log data are maintained in separate locations, by mapping a fixed contiguous size of NVM. Further, the log truncation frequency is set to LLC size (1MB in our case). Separating the log metadata from the log data enable easier traversal of log metadata for retrieval or clean up. When an application developer decides to commit persistent data to NVM, a virtual address is passed, along with a hint of the log granularity. The log record has a transaction ID (TID), the address/offset that points to log data, a pointer to the original data address (not the log data address). The monotonically increasing TID helps with data version conflicts, i.e., conflicts of the log records with the same virtual address are solved using the TID (higher ID number indicates newer log data). For word-based logging, log data and log metadata are similar in size. For object-based logging, the log metadata contains additional fields indicating the size of of the objects logged. Again, key benefits of this hybrid logging approach are that by providing a flexible hybrid interface, (i) the high ratio of log metadata/data is reduced, by avoiding logging every word change, and (ii) high log data costs are reduced, by avoiding logging the entire object even when the data change is less than a word.

Hybrid logging – Experimental evaluation. Similar to the allocator evaluation, we run the same PHT benchmark enabled with two different logging methods: 1.) word-based logging and 2.) the proposed hybrid logging. The X axis in Figure 46 shows increasing numbers of hash operations that would consequently lead to increasing numbers of logging operations. The Y axis, shows the percentage reduction in relative cache misses using the hybrid logging approach. We use the same hash table implementation code as in prior work [146] with a similar transaction interface. For every hashtable key insert, in the case of hybrid logging, 5 word level transactions are replaced by one object transaction, and for a delete operation, 3 word transactions are replaced by one object transaction. This reduces the total NVM writes of 120 bytes (5 log entries with 24 bytes each entry) to 28 bytes (three 8 byte log records pointers (see Figure 44) plus a 4 byte object size field) when adding a new key and value to a hash. The outcome is a reduction of an average of two cache line flushes to one cache line flush. Additional optimizations like flush batching can combine multiple object record updates (approximately 3 log records) to one flush. The data written to log would be the same for both word-based or hybrid logging, as it is application dependent. We observe that irrespective of the number of hash operations, the relative reduction in cache misses with hybrid logging is almost constant, as expected. Additional benefits of hybrid logging can be expected when batching the log metadata flushed. Further, the impact of our optimizations on NVM persistence application was less than 0.7%.

To understand the effectiveness of our mechanisms on other cache efficient applications, we used the persistent B-tree described earlier in section 5.4 with 1.5M operations (same as PHT). Figure 49 shows the performance gains of B-tree using all our optimizations. As expected, gains for cache efficient B-tree was restricted to less than 3.5%, and the average gain was around 1.2%. Most benefits (73%) for B-tree was due to the allocator optimizations and the rest from the page contiguity method with no logging related gains. This is because, each B-tree node was aligned to a word size (we used 8 byte integers as node values), and hence, use of a hybrid approach instead of default word logging was not required. While our current optimizations show less benefits for cache efficient B-tree, our future work would explore more such optimizations.

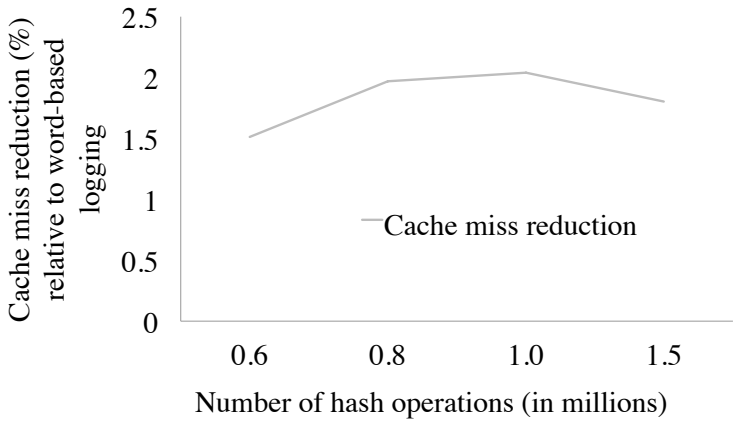
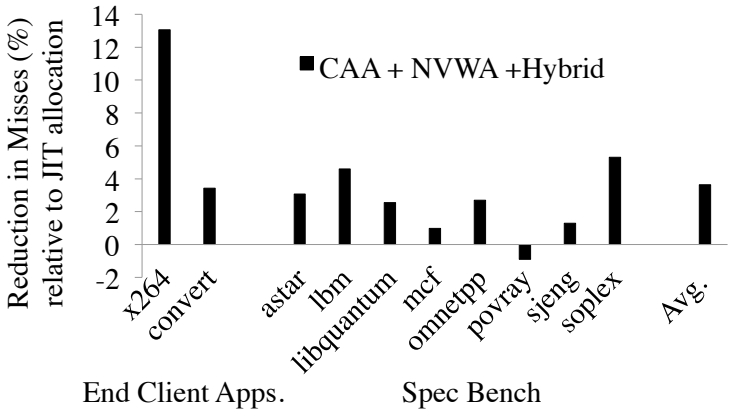


Figure 46: Logging microbenchmark using PHT.

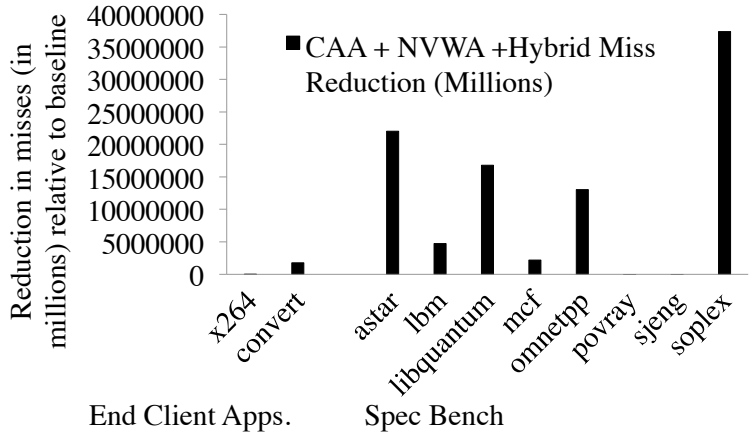
5.4.4 Discussion

With the goal of enabling dual use of NVMs in end user devices, we discussed three optimizations that include page contiguity-based cache partitioning implemented at the OS level, an NVM write-aware allocator with focus on reducing the allocator related NVM updates, thereby reducing the cache impact on NVM capacity applications, and finally, we show the benefits of a hybrid logging mechanism. Each mechanism incrementally improves the average reduction in cache misses (up to 3.6-4%) and up to 12% in some end client applications.

Figure 47a shows the overall effectiveness of combining all three optimizations: page contiguity, write-aware allocator, and finally hybrid logging. In most application benchmarks our proposed mechanisms provide 1-6% benefits and in some benchmarks like x264, the benefits are more than 12%. While in the case of x264, the benefits primarily are from allocator optimizations, in the case of other workloads, the benefits add up from all the three optimizations discussed. In some workloads like 'mcf', we noticed less than 1% improvement. In the case of 'povray', we found the working set of application to be comparatively less, and also most memory allocations by these applications are done at the initial stages, and our optimizations are not effective. Figure 47b shows the total number of misses reduced with CAA+NWAA+Hybrid methods compared to baseline.



(a) Effectiveness



(b) Total miss reduction

Figure 47: CAA+NWAA+Hybrid logging

Simple execution time estimation. Our design optimizations and evaluations specifically focus on methods to reduce the cache misses of NVM capacity and NVM persistence applications. Reducing cache misses reduces the need for direct access to NVM, thus avoiding execution time overheads due to poor NVM write latencies compared to DRAM. Our evaluation shows that for some applications our approach leads to up to 12% reduction is cache misses, where for others the reductions are more modest. Based on prior studies, however, we believe that even 1% decrease in the total cache misses suffered by an application can have a substantial performance impact on the end client applications [66].

Performing an accurate and direct assessment of the impact of our methods on execution time is challenging, however. Currently, PCM devices are not commercially available, and emulating varying NVM read/write latencies in current platforms using DRAM is not possible or accurate. Further, current hardware performance counters for end client devices (Atom) do not provide counters that classify cache misses due reads vs. writes. Hence, to provide a simple back of the envelope estimation, we use PIN-based instrumentation similar to several other prior efforts [56, 146, 111] to capture total the NVM read/write and estimate the runtime impact of our optimizations. We use three models 'Half-Half' - half of the misses reduced by our methods is NVM writes, 'Full writes'- all the reduction is for NVM writes, and finally, one-third of misses are reduced for NVM writes. Using the write latency projections of 1 microsecond from [18] and assuming DRAM read latency for NVM reads, Figure 47b shows a simple projection of the execution time reduction. We stress the fact that, our current scope of work is limited to reducing NVM writes (i.e., cache misses) by software optimization. Also, our estimation does not consider features like out of order execution, parallel issue of read/writes to memory in modern processors and should be viewed as a worst case analysis. As expected, when the number of cache misses due to NVM writes increases (Full writes), our optimizations can provide substantial performance gains for memory intensive benchmarks. Even when the write related misses are substantially less (one third due to writes), our optimization can improve the application execution time by around 6%-8%. Our future work will focus on a more detailed investigation of the execution time impact of our techniques.

5.5 Chapter summary

This chapter analyzes the dual-use of NVM, for memory capacity and persistent storage, mainly focusing on end client devices. Analyzing NVM writes, we find that effective dual-use NVM requires new methods that address cache sharing between persistent and non-persistent applications, in addition to optimizations to the memory subsystem's software stack, including allocators and logging. We develop cache sharing solutions use a novel contiguity-based approach to memory allocation, which reduces by up to 10% the overall

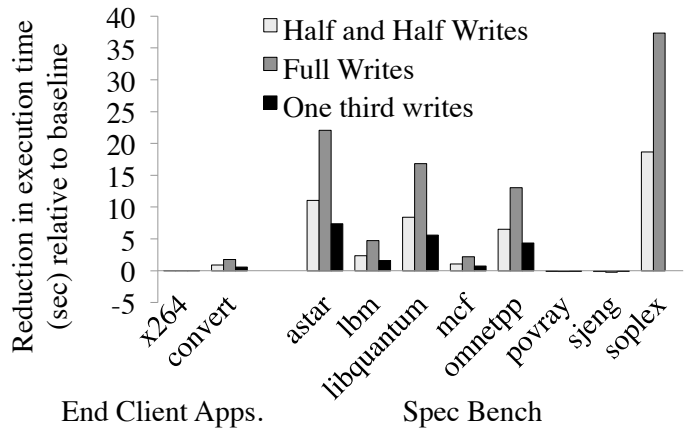


Figure 48: Execution time improvement estimation

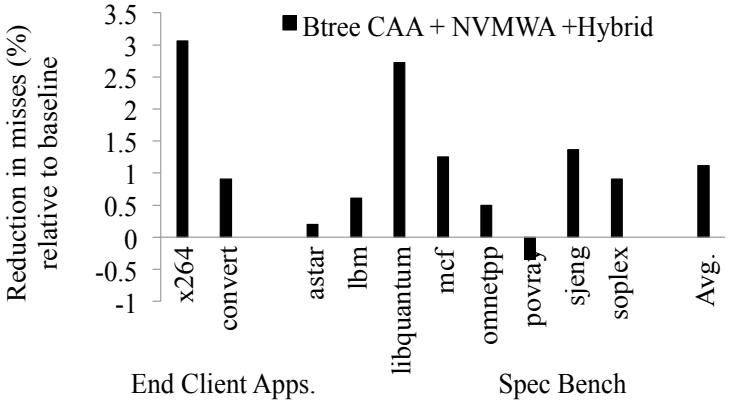


Figure 49: B-tree - CAA+NWAA+Hybrid logging

cache sharing impact experienced by non-persistent applications and caused by persistent co-runners. Further improvements are achieved with an efficient write-aware persistent allocator, leading to reductions in the overall cache misses of up to 12% for client applications and around 4% for SPEC benchmark. Finally, an NVM write-aware hybrid logging approach substantially reduces the NVM writes. An interesting outcome of this research is the experimentally demonstrated need for end-to-end solutions that include optimizing the associativity ways of a cache, modifying the memory allocators used by the operating system and by application libraries, and reducing the cost from providing ACID-based persistence guarantees such as logging.

CHAPTER VI

ENERGY MANAGEMENT FOR HETEROGENEOUS MEMORY

6.1 Introduction

Slow write performance of heterogeneous memory technologies such as NVM is well-known. In the chapter 5, we discussed the problem of cache interference from dual use of NVM, and proposed software-based solutions to reduce the cache interference as well as reduce direct NVM writes arising from the application need for providing ACID-based persistence. Increasing writes to NVM not only slowdown performance but are result in significant energy consumption.

Most prior work has focused mainly in optimizing the performance of ACID-based persistence by either treating NVM as a disk model [118, 67, 57] or NVMs as heap [146, 56, 117] also commonly referred as memory-based persistence (Mem-persist). In Mem-persist, applications use persistent allocators to allocate and update heap objects inside a transaction before committing them. In chapters 3, 4, 5, we discussed several benefits of using Mem-persist over disk-based persistence for a broad range of applications such as key-value stores, persistent in-memory data structures, object-based storage, NoSQL and SQL database. In Mem-persist, for maintaining correctness both the application data and its metadata are flushed from the processor cache to maintain an NVM memory write order, whereas for durability, they are logged. The metadata includes all information required for ACID persistence such as object allocator state and application data log headers. Consequently, providing a strong ACID guarantee not only increases the energy cost due to increase in NVM accesses, but also increases the CPU instructions, hence resulting in a higher CPU energy use. Our analysis shows that ACID-based persistence incurs a CPU energy increase of 7.3x and NVM energy increases of 5.1x compared to a baseline that does not maintain persistence. For computing platforms (such as mobile devices) where energy consumption is a critical factor, it is important that the energy cost of persistence is reduced.

To address this, we propose **energy-aware persistence** (EAP) as an essential element of deploying NVM in end-user devices. Prior work has sought to reduce the Mem-persist ACID performance costs [161, 57, 117, 105] using performance-centric solutions. These solutions either optimize transactions for performance or apply optimizations oblivious of the durability and correctness-related persistent components and hence, are unaware of their energy impacts. In contrast, we (1) analyze the energy implications of the correctness and durability components in detail, (2) identify that durability-related costs are the most significant contributor to persistence energy usage, and quantitatively demonstrate the significance of the CPU-related vs. solely NVM-related cost in the overall energy overheads of supporting ACID. (3) We leverage these observations to incorporate energy-efficient durability alternatives that trade performance or memory capacity in a controlled manner, to meet the constrained energy budgets of modern end-client devices. When such optimizations are insufficient, (4) we propose a a novel relaxed (not delayed) durability model – ACI-RD – that does not affect application correctness when the energy is critically low as illustrated in Figure 50. The figure provides a timing diagram comparing a traditional ACID vs. our ACI-RD design. In the ACID case, after application execution, both application data, and metadata are ordered, flushed and logged, whereas with EAP, when energy is critically low, data durability is relaxed for application data, but not for its metadata, thus maintaining the ACI properties essential for correct operation after a failure. EAP operates with an epoch-based execution model, where the target energy level at the start of each epoch drives the choices of suitable alternatives from (3) and (4) by dynamically measuring the durability-related energy of each ACID stack component.

6.2 Background and related work

ACID-based persistence The processor cache can buffer writes and significantly reduce the impact of high NVM write latency. Current OSes and architectures use a write-back cache update model in which cache lines can be evicted in any order. However, for guaranteeing ACID properties in persistent storage, writes to NVM must be ordered for preserving correctness [117, 105]. For example, committing the metadata before committing the actual

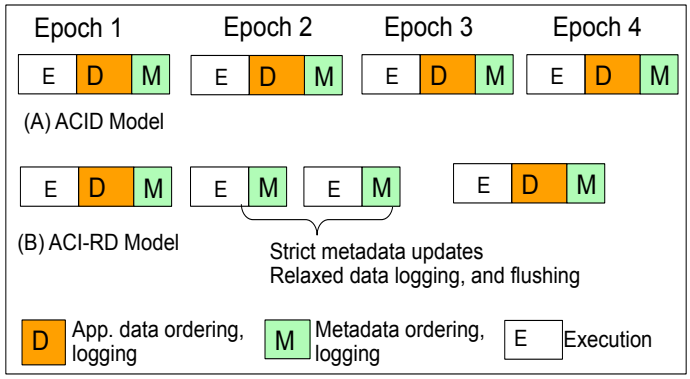


Figure 50: (A) Traditional ACID epoch execution, (B) ACID-RD: Data logging relaxed for critically low energy Epoch 2,3

data can be dangerous. Prior efforts propose using write-through caches [146], using epoch-based cache eviction [57] supported with memory barriers, or introduce a hardware-level persistent cache. In addition, besides the write ordering issues, data durability can be affected due to a power failure that destroys the cached data, resulting in a non-deterministic state. Although applications can flush the cache to reduce non-recoverable failures, to guarantee ACID properties, an additional mechanism such as logging (e.g., undo/redo) is required. However, both flushing the cache and logging are expensive. To reduce these costs, recent works [161, 93, 117, 162, 105] have proposed (i) optimizations to existing persistence mechanisms in hardware and software [161, 93, 162] with a goal of reducing NVM access, and (ii) considered relaxed consistency models [117, 105] that do not impact application performance. We address the energy overheads of memory-based ACID persistence.

NVM memory based persistence (Mem-persist). Recent research on NVM memory-based persistence [146, 56] has explored ACID persistence. Unlike earlier work that treats NVM as a fast disk and uses page-based persistence, these more recent approaches manage persistence at object- or word-size granularity. Mem-persist requires application-level changes to an object to be logged, either with an UNDO log [56], or with a write-ahead log (WAL) [146]. Note that in disk-based persistence, which is used in databases [118, 34], although updates can be smaller than a page, the entire page should be logged. As an illustration, Figure 51 shows the basic Mem-persist operations for inserting a new node in a B-tree inside a transaction. First, a new child node is allocated using the persistent

memory allocator and the allocator metadata (Alloc.C) for this child node is logged. Second, before adding the child node to the parent, the parent is backed into a UNDO log. The backup involves first writing the parent node (P) – considered as an application data – to the UNDO log, followed by a header of the application data (P.hdr) – considered as the application metadata. After UNDO logging, the parent pointer to the child node is updated, and when the transaction successfully commits, the UNDO logs (application data and metadata) are cleared. In the event of a failure before a transaction commit, the application metadata (P.hdr) is used to UNDO any data changes and restore the parent (P) to a state before the child node was added, using the UNDO log. The allocator metadata (Alloc.C) is used for the garbage collection of the allocated memory for the child node. Note that the metadata (Alloc.C and P.hdr) is required for correctness of the application and is typically much smaller (16B) than the data, thus requiring much lower energy costs during updates. We target applications which, as in this example, require memory-based persistence, and we focus on reducing the energy costs associated with providing ACID requirements in Mem-persist.

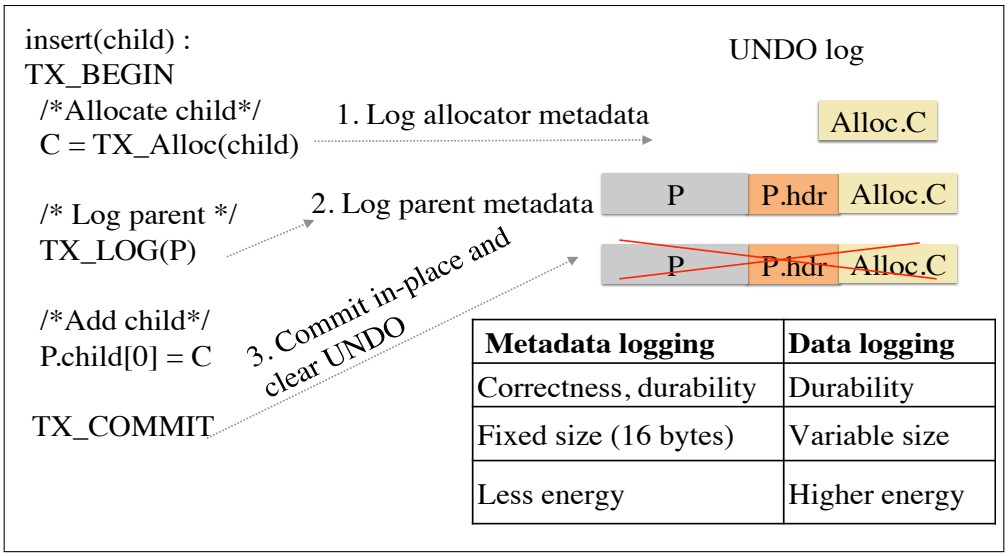
Relaxing persistence. To reduce persistence overheads, Pelly et al. [117] propose a consistency model that allows reordering between one or more independent transactions for better concurrency. Reordering is done only for writes to the cache, whereas writes to the NVM from the cache are ordered. This approach constitutes a variation of the epoch-based consistency model that requires hardware and software changes for reordering writes. Y. Lu et al. [105] propose a loose ordering consistency (LOC) protocol for relaxing intra- and inter-transaction ordering. In their approach, the log area is divided into blocks of 64 bytes with one metadata header for seven blocks. Committing log data in block-groups avoids repeated metadata update. After a failure, the consistency is validated using the group-level metadata. LOC assumes the presence of a nonvolatile cache. Next, [162] improves persistent storage performance by adding intelligence to the memory controller for prioritizing persistent data updates with non-persistent data updates. All the above proposals require hardware and software modification. A recent software-only research, NVRAMDB [118], uses NVM as a disk for page-based persistence, and proposes performance optimizations

specifically for databases. NVRAMDB uses a batched/group commit that first copies the original data to an UNDO rollback/recovery log, then buffers multiple transactions in a DRAM buffer, and finally commits all buffered data to NVM. Although this approach improves the throughput, it does not change the total data logged to NVM. In fact, by buffering transactions in DRAM and then copying to NVM, this solution consumes additional CPU and DRAM energy. In contrast, we propose a group commit update customized for Mem-persist that reduces energy use.

EAP versus prior work. EAP differs from the work reviewed above [117, 105, 162, 118] in several ways. (1) EAP identifies and addresses ACID software components that increase energy. (2) Other than NVM support, EAP does not require additional hardware changes. (3) Prior work principally seeks improvements in application performance, a case in point being the lazy asynchronous, or relaxed atomicity and ordering in [105] and [117]. Using such methods change when certain actions are taken, but they do not reduce the total CPU instructions or NVM accesses. As a result, these proposed methods do not directly reduce the persistence-related energy consumption. (4) EAP's relaxed durability model is designed for Mem-persist where it is important to classify data and metadata for maintaining correctness of heap and application state, unlike [117]. Finally, (5) our mechanisms are driven by the current energy availability, with flexibility to switch between performance and energy-efficient modes.

6.3 Deconstructing ACID energy cost

To understand the energy overheads of persistence, we perform detailed analysis of the correctness and durability ACID components using a simple energy model that considers component-level NVM and CPU energy usage. These analysis results are the basis for EAP's energy-efficient durability principles and ACI-RD design. We focus our analysis for Mem-persist applications that use only NVM [113] as opposed to a DRAM-NVM hybrid model. Our analysis reports the increase in CPU instructions, NVM writes, and CPU and NVM energy. The NVM energy results are for PCM-based NVMs that have 36x higher active write energy than DRAM. Although we use PCM for analysis, the software principles



Metadata logging	Data logging
Correctness, durability	Durability
Fixed size (16 bytes)	Variable size
Less energy	Higher energy

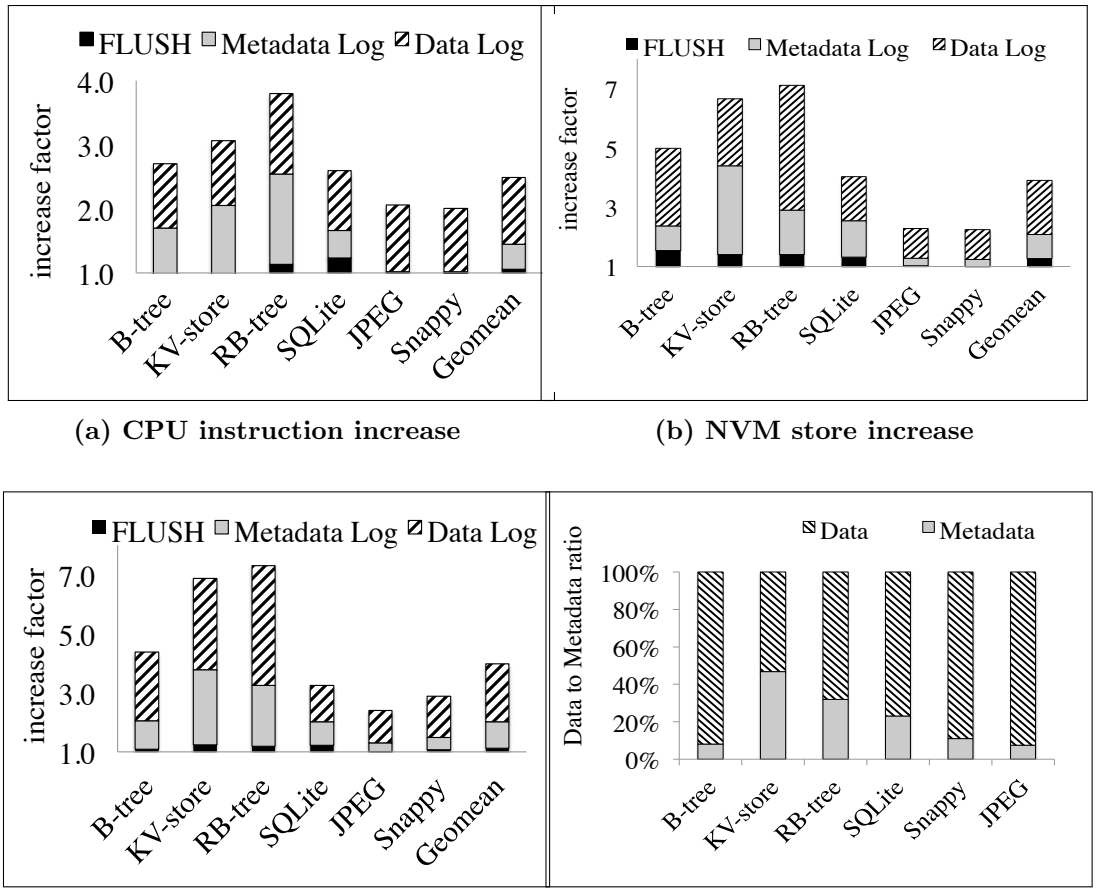
Figure 51: NVM Mem-persist. Shows transactional B-tree child node insertion and corresponding data and metadata logging

are generic and focus toward reducing the necessary parameters such as CPU and NVM use. To validate that our proposal will be beneficial for other competing NVM technologies such as STT-RAM (only 5-10x higher write energy than DRAM), in Section 7.1.5 we run applications in the current system based on DRAM, and measure the system energy using a power meter.

6.3.1 Component-level energy analysis

The ACID requirements of an application are satisfied by multiple user and system-level components. User-level components include the actual application code and the user-level libraries that provide applications with persistent memory allocation/deallocation and transaction support. System-level components can be a persistent filesystem or persistent memory management mechanisms [56, 93], each maintaining ACID properties for their internal, system-level persistent data and metadata structures.

Model. The total energy consumed by an ACID-based application is the sum of energy used by each of these ACID components that order, flush, fence, and log their states. Our energy-aware optimizations, therefore, address these components and their joint operation. Stated more precisely and focusing on the major contributors of energy consumption – CPU and NVM, the following simple equations denote the total energy used by an application,



(a) CPU instruction increase **(b) NVM store increase**

(c) CPU energy increase **(d) Total data vs. metadata log size ratio**

Figure 52: ACID component analysis. Y-axis shows increase factor relative to No-ACID (baseline).

$$\left.\begin{aligned} E_{total} &= E_{APP} + E_{datlog} + E_{metalog} + E_{flush} \\ EA &= E_{datlog} + E_{metalog} + E_{flush} = E_{total} - E_{APP} \\ EA &= EA_{CPU} + EA_{NVM} \end{aligned}\right\} \quad (2)$$

where, E_{APP} denotes energy without ACID, E_{flush} - the energy from cache flush, fence and drain, E_{datlog} - the energy from data logging, $E_{metalog}$ - the energy from metadata logging, and EA - the energy for maintaining ACID guarantees.

NVM emulation and analysis approach. We use an x86-based Haswell desktop system running Linux 3.9.4. The system has 32KB L1 and 4MB LLC write-back cache, Intel

520 120GB flash memory, and 4GB DDR3 DRAM, of which we use 2GB for NVM by mounting the PMFS filesystem [67]. Because byte addressable NVMs such as PCM are not commercially available, to emulate NVM's read and write latency, we inject software delays similar to most prior NVM research [146, 56]. Our emulator periodically gets the total load-store cache misses (every 100ms, using the RDTSCP synchronous instruction as a timer with +-20ns accuracy), and uses the delay model proposed by Dulloor et al. [67] to emulate 100ns load and 400ns store latency. For the CPU and NVM energy we use the RAPL support [81] to first measure the CPU and DRAM energy consumption, and then use the LLC cache misses due to load and store instructions to estimate the NVM energy usage based on the read and write energy values discussed in [98]. In Section 7.1.5 we discuss more details of the dynamic energy estimation. We focus our analysis on fundamental elements necessary for performance and energy such as the increase in CPU instructions and memory access from ACID-based persistent updates. In Section 7.1.5 we show that our analysis and solutions are applicable even when NVM latency and energy cost is same as DRAM.

We extend and optimize the SNIA standard-compatible NVML library from Intel [87] for transactional support. It provides support for transactional object persistence, persistent memory allocations, logging, and persistent barrier. The persistent objects are stored in a large memory mapped region managed by PMFS OS layer. NVML provides an interface for an UNDO log, as shown in Figure 51, and rolls back interrupted transactions. We also extend NVML with WAL-based logging. Further, each persistent object has a different virtual address across restarts but has one unique identifier that is used to load the object from a persistent region.

Applications. Table 14 shows the client-centric persistence benchmarks [105] and applications used in prior studies [105, 93]. (1) B-tree (balance search tree) is a well-known persistence-efficient data structure extensively used in databases, large graph, and even in memory management mechanism [87] due to its O(log n) and O(n) worst case update/search operations and space complexity. Each B-tree node can have many child nodes and a relatively smaller tree depth. The parent and child nodes can fit inside the same cacheline reducing the overall cache misses. However, node additions and deletions require re-balancing

Table 14: Applications for energy analysis.

PersistMem apps	Description	Workload
RB-tree [87]	Cache & memory inefficient self-balancing red-black tree	500K operations with random insert, read, delete operations
B-tree[87]	Persistence-friendly, balance search tree used in databases, S/W cache etc.	Same as RB-tree
KV-store	Simple key-value store built over a persistent hashmap	Same as RB-tree
SQLite [24]	Database used extensively in client devices	500K operations of SQLite benchmark [74]
Snappy [23]	Fast compression library used in chrome and other Google products. We ported it with Mem-persist	1.5 GB directory of image, video, audio, document files
JPEG [9]	Well know image conversion service that we ported with memory-based persistence	40K .jpg files converted to .bmp

and data movement that forces a persistent memory implementation to log these changes. This can significantly increase the cost of persistence. (2) RB-trees (red-black trees) are extensively used in-memory data structures in the OS with O(log n) insert and O(n) worst-case space complexity. RB-trees are persistence and memory-inefficient because every update to a tree node results in flushing and logging both the node and its two child pointer. Next, (3) key-value store (KV-store), like those used as a data cache in both end-user and server platforms, is designed using a persistent hashtable. For adding a KV-store entry, first, a hash entry, a key and a value are allocated using a persistent memory allocator, and the allocations are logged (metadata logging). Second, the data of the key, value and hash entry pointers are updated and logged. Hence, the persistent KV-store also has high ACID cost. Next, (4) the SQLite database widely used in client platforms. It supports both write-ahead logging (WAL), journaling/rollback (referred to as UNDO in this thesis). We use the existing in-memory database and logging feature of SQLite with page-based (as opposed to object-based) persistence to avoid significant code changes to the application. Finally, we use (5) Google-based file/object compression (Snappy) and a popular image conversion (JPEG) used across different OSes.

Analysis. Figures 52.(a)-(c) show the increase in CPU instructions, NVM writes, and CPU energy for applications listed on the x-axis compared to the No-ACID approach that does not offer correctness or durability guarantees. In the stacked bars, FLUSH refers to persistent barrier cost that includes cache flush and memory ordering (fence and drain operations) instructions [141] . Metadata and Data Log refer to the application and allocator metadata and data logging cost, respectively. As the graphs show, supporting ACID for NVM Mem-persist increases CPU instructions, NVM accesses and energy significantly, with durability (metadata and data logging) dominating the cost. The overall CPU energy increases by up to 6.1x for RB-tree of which the data logging alone adds 4.1x overheads. The NVM store increase also shows a similar trend but with even higher overheads. For instance, both RB-tree and KV-store incur 7x more NVM writes. It is important to note that, for KV-store, the metadata to data size ratio is high as shown in Figure 52d. This is mainly due to higher persistent memory allocation and garbage collection cost including the application metadata cost for the reasons we discussed earlier. Therefore, the metadata-related CPU energy and NVM stores increase by 2.5x and 3.72x, respectively. Snappy and JPEG show relatively lower metadata cost as they process less than 50K files, whereas the data logging cost is high from logging large multimedia objects.

6.3.2 Deciphering durability costs

We next analyze the sources of application data and metadata logging costs. We then use the resulting insights to formulate a set of EAP principles for reducing energy usage.

Logging Methods

Prior research on NVM Mem-persist have used either (1) UNDO (refers to journaling) logging [56] or (2) write-ahead logging (WAL) [146]. Also, to exploit byte addressing capability, the Mem-persist research use word or object based logging unlike page-based logging in disk-based systems. We next discuss the energy implications of such logging methods.

UNDO vs. WAL logging for Mem-persist. When using the UNDO logging in a transaction, original data is backed to a journal (log) in the NVM before modifying the data in-place. After a transaction commits successfully, the log is discarded, or else, if the

transaction aborts or if the system fails, the backed up log is used to revert the intermediate updates. Although UNDO requires writing twice to NVM for each transaction (first to UNDO journal, and then to actual data address), it allows in-place writes and read-after-writes. In contrast, WAL reduces the double write bottleneck by appending updates directly to the log, and when the log space runs out, it checkpoints the log contents from the log to the original data location. However, because updates are not in-place, subsequent writes and read-after-writes inside a transaction have to be redirected to a log. To read from the log WAL maintains an index to locate and fetch the latest version of the data in the log. Therefore, WAL offers sequential updates to the log by just appending writes, that can significantly improve performance and concurrency for large multi-core systems with write-intensive workloads. However, for read-intensive workloads, the redirections to the logs can be expensive in terms of CPU instructions and NVM accesses. Additionally, sequential updates are significantly beneficial for disk-based systems, but the gains are limited in memory-based persistent storage [113, 67]. Redirecting every access to a log can result in significant code changes, eliminating the use of the byte-addressable load-store interface.

As discussed by [113, 67], another drawback of WAL for large data updates is that the fixed size log buffers must be frequently truncated and their contents have to be checkpointed to the original data location. Checkpoints require parsing the log records sequentially and copying multiple versions of the same data to the original location. While WAL makes updates faster by only appending to the log, eventually all log entries should be committed, and hence this does not change the total CPU instructions or NVM access. These issues are relevant for page-based logging mechanisms too [114, 55].

Analysis. We analyze the performance and energy impact of UNDO and WAL logging with the B-tree benchmark that uses object-based Mem-persist using the NVML library, and the SQLite that uses page-based persistence. The system setup and NVM emulation are same as discussed earlier. Figure 53 compares the performance (throughput) of WAL and UNDO for SQLite and B-tree. The y-axis is in thousands of operations per second. The x-axis indicates different access pattern. As shown, for small, sequential, write-intensive workload as in the case of SQLite, and All-insert for B-tree, WAL performs marginally

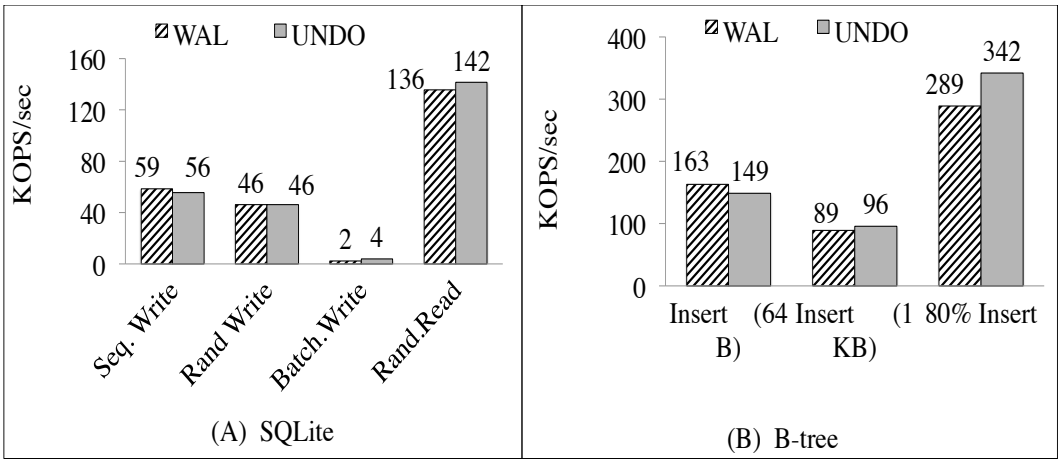


Figure 53: WAL vs. UNDO Kilo OPS/sec for individual access pattern. Y-axis is overhead increase factor relative to No-ACID

better than UNDO. However, for the read-intensive workload (20% writes, 80% reads) and large data updates for B-tree, the UNDO performance is better, and similar results are observed for large SQLite updates. Next, regarding the energy use, Figure 54 shows the increase in CPU instructions, NVM writes, and energy for SQLite and B-tree when using WAL and UNDO. The increase (y-axis) is compared with No-ACID (and no logging), and the values are for an entire benchmark (cumulative) run with different access patterns. We observe that, compared to WAL, UNDO logging reduces CPU instructions, NVM writes and energy usage, for both B-tree and SQLite. *In short, the performance and energy of logging methods vary based on the workload and implementation. Hence, a suitable mechanism that can switch between energy and performance modes is important.*

Metadata Durability Cost

Concerning the energy overheads associated with the metadata persistence, persistent memory allocators can increase the energy use for applications that frequently allocate and deallocate data structures (e.g., key-value stores, B-tree), because such allocations and corresponding data structures are also logged. Prior work [93] proposed NVM write-aware allocators that reduce NVM access by placing complex data structures into DRAM, and just use a more efficient allocation log in NVM. However, for small and frequent NVM allocations, logging and flushing the allocator state updates can become expensive, especially when the *metadata*/*data* ratio increases, resulting in more CPU instructions, NVM

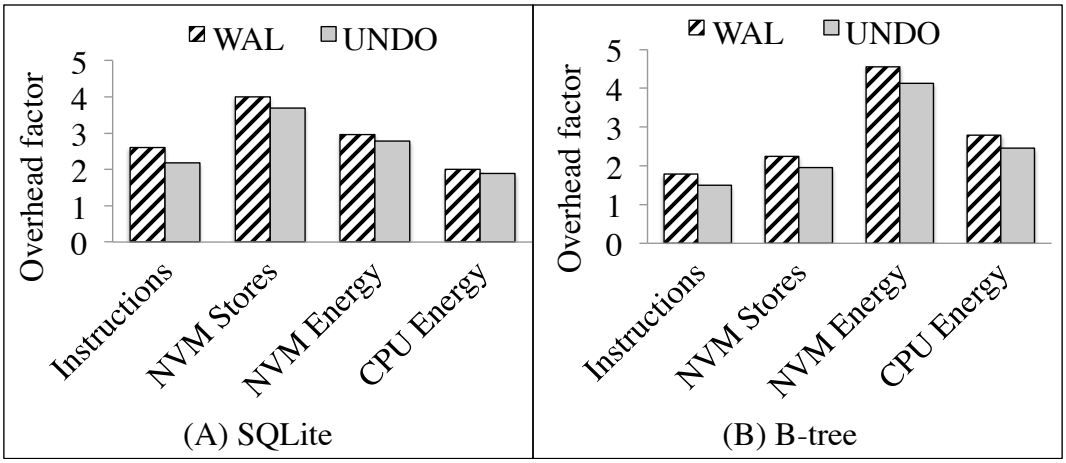


Figure 54: WAL vs.UNDO Instruction, CPU energy, NVM access. Y-axis is overhead increase factor relative to No-ACID

accesses, and metadata durability energy cost.

6.4 Energy efficient ACID principles

Based on the insights gained from the analysis of the ACID overheads in the previous section, we next formulate a set of energy efficient durability (logging) principles. The principles discussed are the first step towards energy reduction under strict but 'not critical' energy budgets for end client devices. In the next section, we propose a relaxed durability model (ACI-RD) for energy critical state.

6.4.1 Flexible logging

As discussed in Section 6.3, although WAL provides marginally higher throughput compared to UNDO for small updates and write intensive workloads, it increases CPU and NVM energy usage for large and read-intensive workloads.

Key idea. Motivated by these observations, EAP provides a dynamic logging mechanism that transparently switches to an energy-aware logging mode (WAL to UNDO, and vice versa). When energy is not a constraint, then applications start with WAL as a default mode. At fixed time intervals – epochs, the logging library measures the available energy budget. When energy availability becomes limited, the library switches to UNDO logging.

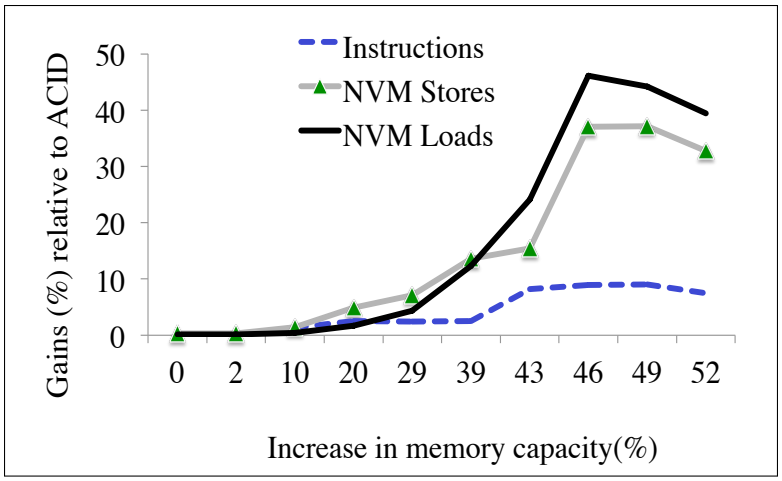


Figure 55: B-tree: y-axis shows gains relative to ACID by trading memory capacity (x-axis in %).

The dynamic switch from WAL to UNDO is initiated only for new transactions since splitting the log for a dirty and uncommitted transaction across WAL and UNDO is suboptimal for logging and recovery. Additionally, all the pending WAL transactions (including nested transactions) are committed before the switch. These restrictions simplify the extensions required in SQLite to incorporate EAP. We evaluate the overheads of frequent switching (due to small epochs) in Section 7.1.5.

6.4.2 Gain energy by trading capacity

We next discuss the principles for reducing persistent memory management overheads by trading capacity when energy availability is limited.

Reduce allocator work. Modern DRAM and persistent memory allocators strive to provide fast allocation and reduce memory fragmentation. They maintain memory object slabs of different sizes (in powers of two), and service request from such slabs. To align requests to the nearest slab, allocators perform complex operations such as merging multiple small objects. When ineffective, they request from the OS (via *mmap()* or *break()*) to allocate a batch of pages (64K - 2MB), and these operations consume significant CPU energy. Requests for smaller batches result in frequent OS requests, whereas larger batches impact free memory capacity due to fragmentation. The overheads of persistent memory allocators are even higher since the allocator state must also be persisted. Hence, to reduce

Table 15: SQLite gains from trading 35% higher capacity.

Instructions	NVM Loads	NVM Stores
11.86%	9.28%	10.68%

energy use (and instructions and NVM accesses), EAP's energy efficient durability trades off capacity with large OS allocations (64MB) only when the energy budget is a constraint. Also, complex merge operations are avoided to reduced CPU work and energy.

Reduce garbage collection overheads. More than memory allocation, the cost of persistent memory garbage collection is even more significant. Most garbage collection methods use a 'mark and sweep' approach (first mark objects and then delete). Prior research have analyzed the performance overheads of DRAM-based garbage collection in end-user devices [116] and server machines [85]. For NVM, the allocator-related overheads are even more substantial since the allocator should also persist (and log) deletions before requesting that the OS release memory using the expensive *munmap()* call. Hence, when the energy is a constraint, EAP trades capacity to reduce energy consumption by delaying the garbage collection, without affecting the correctness. We modify the allocator to mark the objects for deletion but free them only when the available free NVM capacity is below a threshold (currently, we use the OS swap threshold). We extend this to the OS-level garbage collection for delaying the release of application heap pages by adding and setting a special one-bit page flag.

Analysis. Figure 55 shows the combined effects of energy efficient allocation and delayed garbage collection. The results show reduced NVM load-store accesses and CPU instructions compared to the ACID-based approach (y-axis in %). The x-axis shows the increase in NVM usage as a trade-off for energy. Trading off capacity reduces allocator and garbage collection cost thereby reducing CPU instructions and NVM access. It is important to note that the reduction in NVM accesses is higher compared to the decrease in CPU instructions because recycling objects require several expensive *memset()*, *memcopy()*, and FLUSH operations. Beyond 45% increase in the capacity, the gains reduce because after reaching the swap

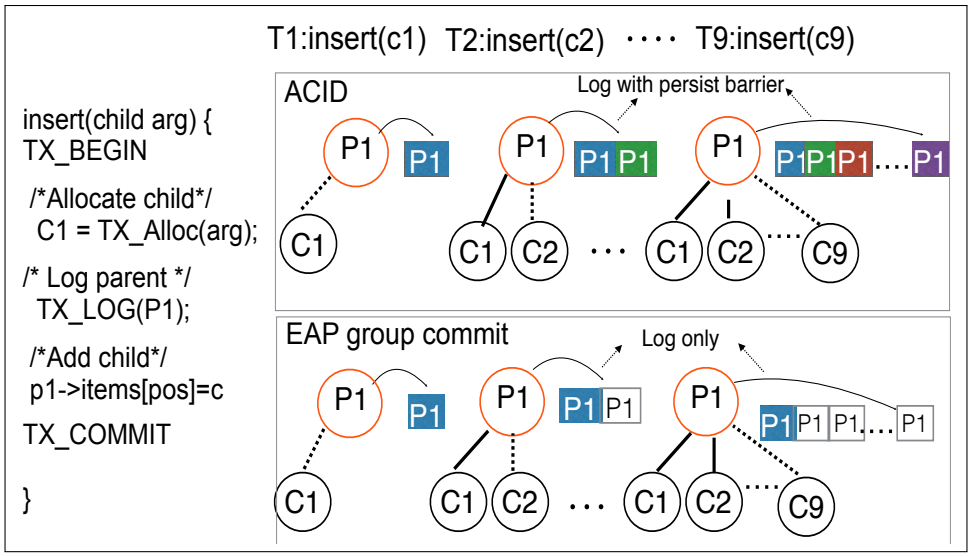


Figure 56: Mem-persist group commit. Circles P,C represent B-tree parent, child nodes. Shaded and non-shaded P1 in square indicate undo-log w/o persistence barriers

threshold, both the persistent memory allocator and the OS have to aggressively release memory. These actions increase the overall work done by both the allocator and the OS, thereby increasing the total CPU instructions and NVM accesses. Furthermore, a subtle but important reason is that the library allocator maintains all objects as nodes in a B-tree, and their lookup or update time increases as we delay the garbage collection. We observe similar trends for SQLite (see Table 15), but the benefits are lower due to a custom page-management.

6.4.3 Memory persistence group commit

Transactional updates with persistence barriers can be very expensive. Grouping smaller transactions into a larger one, referred to as group commit protocol in databases, is well-known. NVRAMDB mentioned in Section 6.2, uses a group commit protocol that buffers updates in DRAM and lazily commits logs and data to NVM [118]. However, NVRAMDB is performance-centric and lacks energy awareness. It increases CPU instructions and accesses to DRAM and NVM by adding one more DRAM buffer. Instead, we propose an energy-aware group commit protocol for Mem-persist.

Key idea: In a traditional ACID design, a persistence barrier (FLUSH for application

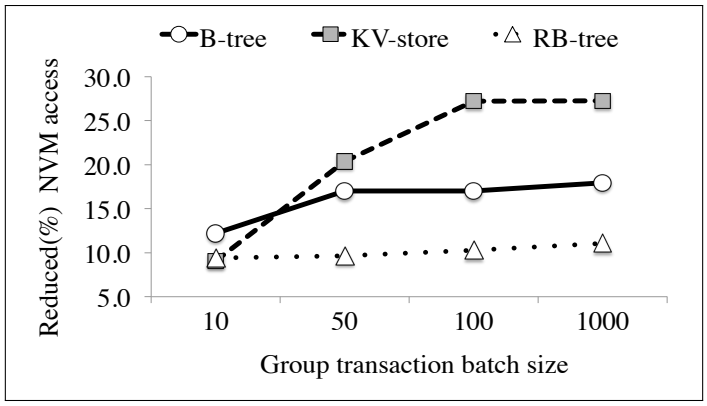


Figure 57: Impact of group commit batch size. Higher object re-access reduces NVM access

data including log [118]) is applied twice, once to the UNDO log before an update, and once to the original data after an in-place update. In contrast, in EAP, we apply barriers only twice in a group of transactions instead of every transaction. The first barrier is used when an object is modified for the first time a group and the next barrier when the entire group commits. Also, unlike NVRAMDB [118], our design does not buffer the log updates in DRAM but creates a clean, separate undo log when an object is updated for the first time in a transaction group. For subsequent transactions, logging and in-place updates happen without persistence barriers. Figure 56 shows the pseudocode for inserting a node in B-tree. Before the parent node pointers are updated, a UNDO log is created for every child (update or delete operations). For a transaction batch size 9 in this example, the persistence barrier for parent node (P1) is only applied for T1 and T9. If a failure or abort occurs before the group commit, all the updates inside a batch are reverted using the UNDO log. This guarantee of "all or none" is same as the prior group commit approach. We implement this by adding a single bit to the object's allocator metadata and setting the flag first time an object is modified inside a group and resetting the flag when the entire group commits. Figure 57 shows the impact of transaction batch size in the x-axis. Intuitively, the benefits are higher for objects that are reaccessed in the same group. B-tree benefits by reducing the redundant barriers for the parent node when multiple new child nodes are inserted, whereas in KV-store repeated barriers for parent hashtable structure is avoided. RB-tree with less reaccess has minimal impact. Hence limiting group transaction size to increase reaccess per

batch is important.

Atomic commits. New persistent memory-specific x86 atomic instructions such as CLWB for cache-line write back without invalidation but ordered store fences, PCOMMIT for non-temporal stores, and CLFLUSHOPT for optimized cache-line flush [86] can reduce CPU instruction and NVM access cost by avoiding the need to log values smaller than 64 bytes. This can significantly reduce metadata logging cost and data logging cost for updates smaller than 64 bytes.

6.5 Relaxed durability with ACI-RD

When the energy budget is critically low, the efficient durability principles discussed in the previous section are not sufficient. In other words, always using an ACID approach in end-user devices can substantially drain the battery power preventing an application from running to completion. This can impact not only ACID-based but also other non-ACID/background applications too.

Key Idea. In a strict ACID-model, all application state that includes object data, its metadata (object headers), and library metadata (persistent memory allocator state) are logged to a consistent log. In the group commit approach discussed earlier, the metadata and data logs are written for each transaction, and only the persistence barriers are relaxed in a group. For critical energy states, when such optimizations are insufficient, we propose a relaxed durability model – ACI-RD. ACI-RD reduces the frequency/increases the interval between application data logging, but without delaying the associated metadata (application and library metadata) logging critical for correctness. When the durability is in the relaxed mode, the metadata is updated to a separate UNDO log. Further, an application can transition from a strict ACID phase to ACI-RD, and vice versa, depending on the available energy. These fixed time intervals, when either ACID or ACID-RD is used, are referred to as 'epochs'.

Why to use a separate ACI-RD metadata UNDO log? The metadata is critical for correctness because in the event of a failure during an ACI-RD epoch (the interval when the application data is not made durable), all metadata updates in the UNDO log are used

to revert the changes, and restore the last checkpoint state where both data and metadata were made durable in a consistent log. Specifically, in a Mem-persist logging, where the UNDO log contains both application- and library-related metadata, the UNDO log provides information to garbage collect the memory allocated in an ACI-RD epoch. Further, it clearly segregates the consistent updates before an ACI-RD epoch from the relaxed updates during ACI-RD. All changes during ACI-RD can be reverted, and just using one log will increase the complexity of sequentially parsing the entire log and classifying consistent and ACI-RD updates.

Transition from ACI-RD to ACID. During the transition to ACID, EAP first enables load and store fences, flushes all data updates from the cache-lines, followed by all metadata updates, and finally issues a load-store fence, so as to guarantee all updates from the previous phase complete. This provides the same correctness guarantees as ACID. In-flight errors can exist in both ACID or ACI-RD, and can be avoided if the future hardware provides an acknowledgment, as proposed by [47].

ACID-RD Steps. Figure 58 shows the update and recovery steps for two variables A, B in both ACID and ACI-RD epochs. When the variables A and B are updated with values a_1 and b_1 in an ACID epoch with no energy constraints in ①, both data (a_1, b_1), and the metadata (address of A, B, and size of update) are written to a consistent log. When the energy becomes a constraint ②, ACI-RD is enabled. In this case, for updates of the variables A and B with values a_2 and b_2 only the metadata (A, B address, and update size) is written to the ACI-RD UNDO log. Note that, the data is updated in place with FLUSH or atomic updates, if applicable. If a failure happens as in ③, first the updates to A and B are reverted using the metadata log as shown in ④, and then the consistent datalog is used to restore the previous values a_1 and b_1. The result of this mechanism is a trade-off between durability (D in ACID) and energy, without compromising application correctness.

Intuitively, as the ACI-RD epoch time interval increases, the size of the data not made durable increases. Hence, a failure during the ACI-RD epoch increases restart cost. More formally, as shown in Equation 3, after a failure in the ACI-RD epoch, the restart time (R_{ACI-RD_t}) is approximately the sum of the time to undo all updates in the ACI-RD

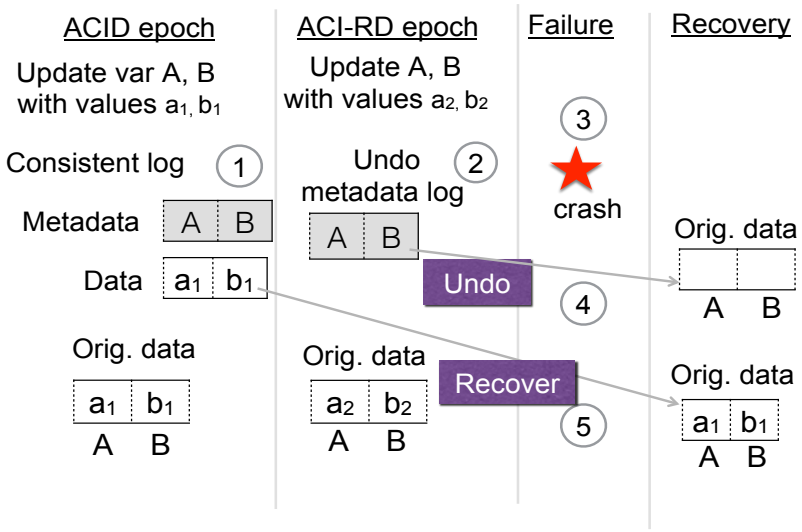


Figure 58: ACI-RD steps, ACI-RD epoch is energy critical state.

epoch ($UNDOt$), the time to recover data from the consistent log, and the time to re-execute the 'lost' transactions. The time to recover from a consistent log is equal to the ACID-based restart time (R_{ACID_t}), whereas the time to re-execute the 'lost' transactions is the product of number of transactions relaxed ($N_{Trans_{epoch}}$) and the average time per-transaction($Trans_t$). $UNDOt$ directly depends on $N_{Trans_{epoch}}$, and average UNDO time per transaction ($UNDO_{Trans_t}$).

$$\left.\begin{aligned} R_{ACI-RD_t} &\approx R_{ACID_t} + UNDO_t + (N_{Trans_{epoch}} * Trans_t) \\ UNDO_t &\approx N_{Trans_{epoch}} * UNDO_{Trans_t} \end{aligned}\right\} \quad (3)$$

To implement the ACI-RD mechanism, given an energy budget E, the application execution is divided into per-epoch intervals of time t_{msec} with per-epoch energy budget E_{budget}. We assume the value of E_{budget} is known and is equal to the per-epoch energy of metadata only approach which we identify by sampling an epoch at runtime [58]. After the end of each epoch, EAP estimates the increase in the epoch's energy usage (ΔE_{epoch}) and the increase in the number of transactions ($\Delta Trans_{epoch}$) relative to current and previous epoch transactions $Trans_{epoch_i}$ and $Trans_{epoch_{i-1}}$. As discussed in Section 6.3, the total energy for persistent applications is a factor of application execution, data logging, cache flush, and metadata transactions. The choice of using this simple estimation is to avoid overheads

```
for each epoch do
    if energy_save_mode = true then
        /*EAP-ACID (efficient durability)*/
        Switch to undo logging
        Apply EAP batch allocation
        Apply EAP delayed garbage collection
        Apply group commit transactions
        Find ΔTrans_epoch from Equation 4
        if ΔTrans_epoch <= 0 then
            continue;
        end
        if energy_critical = true then
            if commit.size < cacheline.size then
                atomic commit;
            end
            if ΔTrans_epoch > 0 or commit.size > maxsize then
                /* ACI-RD epoch */
                Apply fence
                Update data in-place, FLUSH, drain
                Log metadata to UNDO log
                For transaction commit, return special code
            end
        end
    end
end
```

Algorithm 2: EAP efficient durability and ACI-RD steps.

(including energy) from use of a more complex model.

$$\left.\begin{aligned} \Delta Trans_{epoch} &= Trans_{epoch_i} - Trans_{epoch_{i-1}} \\ \Delta E_{epoch} &= (E_{epoch_i} - E_{budget})/E_{budget} \end{aligned}\right\} \tag{4}$$

EAP Execution. Algorithm 2 shows EAP's sequence of steps for reducing energy usage. EAP's efficient and relaxed durability are activated only when the energy saving mode is enabled – a feature available in most end-user devices. When the energy budget is low, but not critical, EAP enables the energy efficient durability principles described in Section 7.1.3 – switching from WAL to UNDO logging, group commit, batched allocation, and relaxed garbage collection – while still maintaining ACID guarantees. We refer to this mode as *EAP-ACID*. In the next epoch, the EAP runtime checks if EAP-ACID is sufficient to meet the per-epoch energy budget, and if not, ACI-RD is activated. These steps are repeated for subsequent epoches until the energy budget is met. Note that unlike checkpointing/recovery protocols with fixed checkpoint interval, in a transactional application, the logging frequency

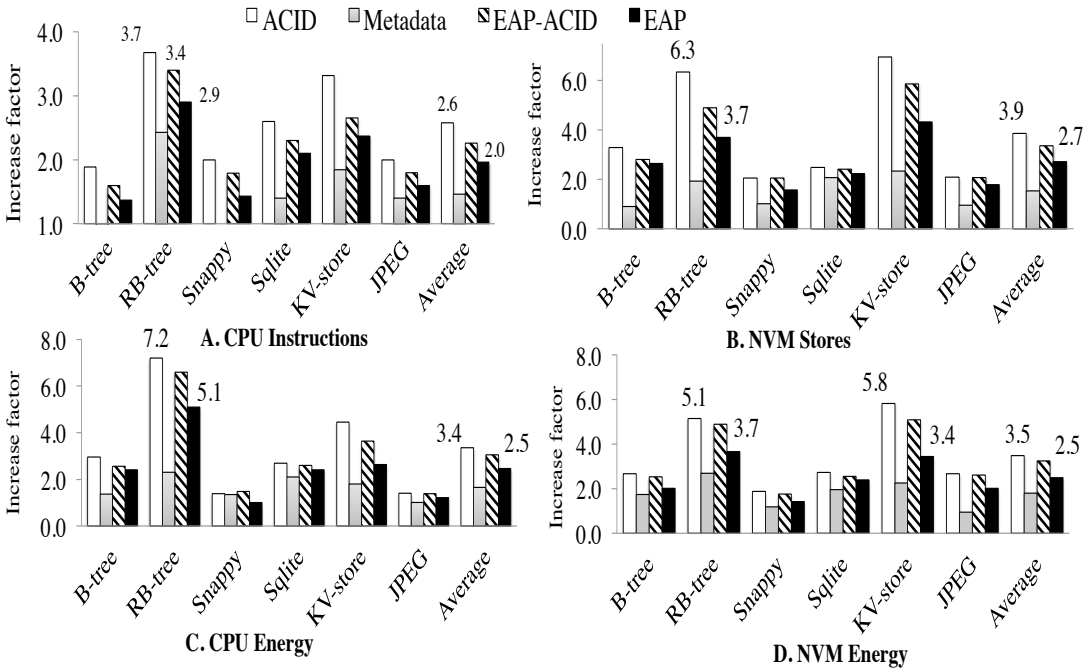


Figure 59: EAP impact:Y-axis shows increase factor(x) relative to FLUSH

depends on the number of transactions executed by the application. EAP uses the energy budget to tailor/relax logging, while still maintaining correctness. When ACI-RD is used, atomic commits can be used for the metadata log, as shown in Algorithm 2. Further, durability for objects larger than a threshold size is relaxed as they consume significantly higher energy.

6.6 EAP Evaluation

The key goal of the evaluation is to understand the impact of the proposed energy efficient durability (EAP-ACID), relaxed durability (ACI-RD), and the overall benefits of EAP that combine EAP-ACID and ACI-RD. We also analyze the implications of ACI-RD on restart time after a failure, and the overall impact on the system energy.

Methodology. In Section 6.3, we described our experimental platform details, NVM emulation with software delay, and the use of the RAPL [8] hardware counters to estimate the NVM and CPU energy. The energy counters are queried dynamically with the minimum possible frequency of 100ms to estimate the increase in ACID-related instructions, NVM accesses, and CPU and NVM energy. Our evaluations focus on the relative energy increase

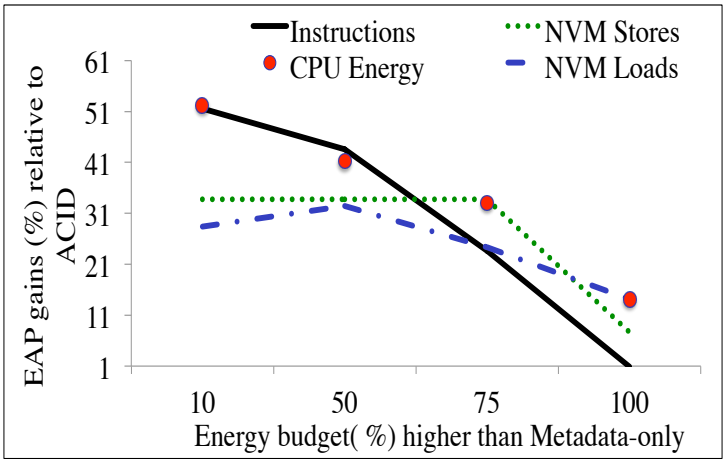


Figure 60: Impact of energy budget

because of ACID, and the evaluation trends are relevant for other NVM technologies.

Baselines. We evaluate EAP, which combines EAP-ACID and ACI-RD, by comparing with following persistence methods. ① FLUSH (baseline) just flushes, fences, and orders data in the cache without satisfying durability and correctness. ② Metadata-only approach guarantees only correctness without data durability by combining FLUSH and logging only the metadata. This approach is not useful for most transactional applications. ③ EAP-ACID provides efficient durability via flexible logging, energy-aware allocation and garbage collection and the group commit mechanism.

6.6.1 Reduced energy use with EAP

Figures 59.(A)-(D) analyze the implications of persistence on increase in CPU instructions, NVM writes, CPU energy, and NVM energy, respectively. The y-axis shows the overheads as factor increase relative to the FLUSH approach as a baseline. The experiments use the applications and benchmarks introduced in Section 6.3. For this analysis, the applications runtime vary between 28 to 64 seconds. In EAP-ACID and EAP, the epoch time is set to 450ms, and we later discuss the epoch interval sensitivity. All tree data structures – B-tree, RB-tree, and KV-store perform insert, find, delete and overwrite operations with 128 byte values.

Analysis: Although B-tree is persistence friendly, ACID increases CPU-instruction by 1.9x and NVM accesses by 3.28x compared to the baseline. EAP-ACID with its energy

efficient optimization reduces instructions and NVM accesses by 30% and 48% respectively compared to ACID. These benefits are mainly from the Mem-Persist-based group commit that reduces multiple persistent barriers for the same parent when child nodes are inserted, and trading capacity reduces the overhead of garbage collection. In contrast, EAP, by using ACI-RD with 400ms epoch intervals further reduces CPU and NVM energy by 61% and 67%, respectively. RB-tree, a persistence inefficient benchmark, has the highest ACID cost. Avoiding a log update for one node avoids the logging cost for two other (child) nodes. While EAP-ACID shows minimal gains (28% lesser instructions), in contrast, EAP reduces instructions and NVM access by 80% and 2.6x, respectively. The Metadata-only approach offers higher gains, but is less useful. For KV-store, EAP reduces the high CPU and NVM energy costs of ACID by 2x and 2.5x, respectively. EAP still exhibits high overheads because of the allocator cost. Redesigning the KV-store data structure can reduce such cost. For Snappy [23], which uses an input workload of 50K files (total 1.5 GB), and file sizes that vary between 10KB-140MB, the metadata logging cost is negligible. However, persisting large files with FLUSH is expensive, and the data logging is even more expensive. EAP relaxes logging for large data updates (see Algorithm 2), which provides considerable improvement. For applications such as Snappy, strict ACID is unnecessary because only files for which compression failed require re-compression. EAP provides a transparent support without depending on the application developer. JPEG image conversion [9] shows similar trends with only 40K transactions, but incurs higher NVM store access because the output files can be larger (up to 30%) adding to the logging cost. With SQLite, EAP-ACID's flexible logging and allocator optimization reduce instructions (16%), and NVM writes (12%) compared to ACID with WAL-based logging. ACI-RD reduces CPU and NVM energy by additional 26-28%. Furthermore, using a page-based persistence limits the gain [114], and redesigning SQLite for Mem-persist in the future can improve the benefits.

Summary and Discussion. In summary, EAP, by combining EAP-ACID and ACI-RD provides up to 2.4x reduction in NVM energy and 2.1x reduction in CPU energy (RB-tree) for data structures that are not persistence friendly, and up to 64% and 40% CPU and NVM energy gains for persistence-efficient applications (B-tree). Although EAP-ACID is

beneficial, for critical energy conditions, ACI-RD is required to run applications completion. EAP does not compromise correctness. Support for multi-cacheline commits and hardware-based energy profiling can reduce metadata and software energy cost.

6.7 Chapter summary

In this chapter, we analyze the energy overheads of persistence, specifically identifying durability costs as the most significant contributor to energy usage. To reduce durability cost, we propose energy-aware persistence (EAP) that first trades-off a small fraction of performance with a flexible logging, NVM capacity with delayed garbage collection, and applies NVM group commit method under low-energy budgets. For critical energy conditions EAP relaxes durability (ACI-RD), thereby reducing energy significantly without impacting correctness. EAP's evaluation using benchmarks and applications shows up to 2.1x CPU energy and 2.4x NVM energy reduction supported by reduced CPU instructions, NVM access, and energy use compared to a strict ACID approach. An interesting outcome of this research is that, to reduce persistence cost, apart from NVM energy, it is important to reduce the CPU energy.

CHAPTER VII

APPLICATION SUPPORT FOR HETEROGENEOUS MEMORY

The previous chapters discussed the system support for heterogeneous memory and including OS-level optimizations required to facilitate the application-transparent use and also achieve performance benefits. Apart from the OS-level support, application and service-level redesign for heterogeneous memory are necessary for achieving optimal performance and reliability guarantees. In this chapter, we discuss the application and service-level redesign by considering applications that are widely used in high-performance computing, cloud, and end-user environments. First, we discuss optimizing the performance of checkpoint-restart using byte-addressable NVMs, second, we discuss methods to improve the storage performance for web browser-based applications running in a sandboxed environment. Finally, we discuss the redesign of LSM-based NoSQL database for achieving higher parallelism resulting in performance gains.

7.1 Checkpoint restart for HPC

Moving toward the exascale, the failure rate of applications is expected to increase to the order of tens of minutes. For overcoming the lack of application availability, there is an urgent need for robust fault tolerance mechanisms with low impact on application performance. Checkpoint/restart is a well-known and widely used fault tolerance technique used in a majority of current HPC applications. A checkpoint is a snapshot of application state stored in persistent devices that can be used for restarting execution after failure. However, with increasing problem sizes and failure rates, checkpoint sizes per processor are increasing at a higher rate than available I/O bandwidth [39], leading recent studies to conclude that for exascale machines, $checkpointsize/IObandwidth$ ratio must be drastically reduced to make such systems usable [138].

Prior research has pointed out that traditional checkpoint methods using parallel file systems (PFS) do not scale mainly because of problems that include limited I/O bandwidth

and contention at the I/O subsystem-level which is already apparent on current petascale machines [37]. New I/O methods using intermediate staging nodes for buffering checkpoints can insulate applications from slow disk latencies [29, 164], but researchers are exploring additional ways to reduce I/O data volumes and reduce necessary data movements [29, 95, 89].

This chapter explores NVM-checkpoint, a promising alternative to disk-based checkpointing by adapting and improving multi-level checkpointing methods that have already been shown to exhibit 30-40% benefits over traditional PFS-based checkpointing methods [110]. Multi-level checkpoint combines frequent local storage checkpoints with less frequent remote checkpoints (e.g., to neighboring or I/O staging nodes). Local checkpoints can reduce the resources consumed by checkpoint data movement (in terms of data volume and compute node CPU), as well as the contention for resources such movements may impose on application processes. Another argument for these methods is that a substantial portion of application failures is due to soft errors, recoverable via local node reboot or application process restart, vs. hard errors that render the entire compute node unusable. In fact, recent results [62] show that in the ASCI Q machine at LANL, about 64% of failures are due to soft errors, and hence can be handled with frequent local checkpoints. Another study from Intel [43] points out that soft error rates can increase by up to 32x moving towards 16nm from the 90nm chip fabrication process. A 64-128 core per node configuration and multiple sockets (say, five sockets) can result in a soft error rate increase of up to 100-120%. Using node local checkpoints permits each node to restart from its local state without the need to access the distributed storage infrastructure.

The NVM-checkpoint described in this chapter uses NVM as a virtual memory and utilizes several key hardware features, including byte addressability, hardware-supported virtual addressing, processor caches, and the ability to use NVM as an extension to DRAM when this is a scarce resource. To overcome the bandwidth limitations of future NVMs, it exploits hardware paging support and page protection techniques, as well as a novel pre-copy checkpoint method. This pre-copy method starts moving checkpoint data even before a synchronous checkpoint is initiated, thereby reducing the total data volume to be moved

at checkpoint time. Further, by providing a remote memory access interface to applications, NVM-checkpoints extend this pre-copy scheme to remote checkpointing, as well, leveraging the RDMA nature of the interconnects used in high-end machines. Here, pre-copying reduces peak bandwidth usage, which then reduces the likelihood of interconnect contention. NVM-checkpointing is fully implemented on Linux-based nodes and is evaluated with three well-known HPC applications. Experimental results demonstrate 15% reduction in local checkpoint time compared to RAMDisk of which 8-10% is due to pre-copy, up to 46% reduced peak interconnect usage using the remote pre-copy mechanism, and 40% improvement in application execution time compared to an asynchronous (non-blocking) approach without pre-copy.

7.1.1 Checkpoint-restart background and motivation

Application-initiated checkpointing. Checkpointing mechanisms differ based on the level of application transparency, synchronization, and storage hierarchy. Regarding transparency, they can be broadly classified into application-initiated vs. transparent checkpoints. Transparent checkpoints do not require applications to handle failures explicitly by saving an entire process address space. In contrast, application-initiated checkpoints store only those data structures identified by application developers. When the application footprint is large, transparent mechanisms incur high storage cost and space. Studies show that large-scale HPC applications mostly use application-initiated mechanisms [38] and require less storage space. This thesis scope is, therefore restricted to application-initiated checkpoints.

Multilevel checkpointing. Next, considering the storage hierarchy, checkpoint data can be stored in a globally distributed storage using PFS (e.g., Lustre), or to an intermediate staging I/O node (e.g., using I/O methods like ADIOS [29]), or using a hybrid multilevel checkpoint approach. Multilevel checkpoint designs address the limitations of traditional PFS-based approach by storing checkpoint data at multiple locations, from local scratch memory to storage resources (i.e., RAMDisk or SSD or disk) at remote neighbors (i.e.,

Table 16: NVM vs. DRAM H/W performance [18].

Attributes	DRAM	PCM
Write Bandwidth	$\tilde{8}$ GB/Sec	$\tilde{2}$ GB/Sec
Page Write latency	$\tilde{2}$0-50 ns	$\tilde{1}$ us
Page Read latency	$\tilde{2}$0-50 ns	$\tilde{5}$0 ns

peers) or designated (i.e., I/O) nodes, and finally to the PFS. The rationale is that application reliability will improve with increased levels of data redundancy. Studies have shown that both local and remote node checkpointing scale well and reduce data movement cost. However, an issue with this approach is that they rely on local memory to achieve higher performance, but future machines are already predicted to be memory scarce. Therefore, NVM-checkpointing uses NVMs as node-local stores, and it overcomes the NVMs' potential bandwidth limitation (around 2 GB/sec, see Table 16) with novel pre-copy based methods that alleviate bandwidth pressure. Section 7.1.3 describes these mechanisms in more detail.

NVM for checkpointing. As discussed extensively in this dissertation, the hardware limitations of NVM such as 10x slower writes, 5x lower bandwidth, two orders of lower write durability, and 40 times higher write energy/ bit are well-known. In contrast to prior NVM checkpointing research that treats NVM as a disk, in the dissertation, we use NVM as a hybrid memory connected via. memory controller. However, NVM is not directly exposed to applications for checkpoint data writes. Instead, applications use specialized NVM interfaces in a similar manner as with explicit application-initiated checkpoint operations in HPC codes, to provide information regarding the data to be placed in NVM, i.e., checkpointed data. During computation, application data remains in DRAM, thereby avoiding potentially substantial application slowdowns (shown to be up to 25% [99] for certain classes of write-intensive HPC codes). Also, we use our OS-level support discussed in Chapter 3 for efficient NVM management and data movement across the DRAM/NVM boundary. Our NVM checkpointing mechanism specifically takes advantage of NVM byte addressability and hardware support for management. To deal with limitations such as limited bandwidth and slow writes to NVM, it uses mechanisms like pre-copy and shadow buffering.

Table 17: Checkpoint model notations.

T_{total}	total execution time
$T_{compute}$	total compute only time
$T_{restart}$	total checkpoint fetch and restart time
T_{lcl}	total local checkpoint time
O_{rmt}	overhead due to remote checkpoint
I	Checkpoint interval time
t_{lcl}	local checkpoint time
t_{rmt}	remote checkpoint time
R_{lcl}	local checkpoint fetch time
R_{rmt}	remote checkpoint fetch time
$MTBF$	Mean time between failures($1/\lambda$)
$MTBF_{lcl}$	failures recoverable from local nodes
$MTBF_{rmt}$	failures requiring remote node recovery
F_{lcl}	number of failures recoverable from local node
F_{rmt}	number of failures requiring remote node data
$NVMBW_{core}$	effective NVM bandwidth per-core

7.1.2 Performance model and goal

The performance of a checkpoint mechanism depends on several factors such as MTBF (mean time between failures), checkpoint data size of application, storage hierarchy, and the hardware devices used. The goal of our work is to understand the benefits and implications of NVM on checkpointing performance. Therefore, we extend an existing two-level checkpoint model [165] to suit our NVM based study.

Figure 61 shows a basic timing diagram of a multilevel checkpoint, with remote checkpoint overlapped with the next computational and local checkpoint step, as commonly done with asynchronous non-blocking I/O operations. The total application execution time is directly impacted by the performance of the local checkpoint, remote checkpoint, and the restart/recovery performance. The total runtime of an application can be denoted by

$T_{total} = T_{compute} + T_{lcl} + O_{rmt} + T_{restart} + T_{recomp}$ (1)

where T_{recomp} is the computation after the last checkpoint before application failure. This is typically wasted computation that needs to be re-executed due to failure.

Improving local checkpoint. The local checkpoint performance is dependent on the

total number of local checkpoints (N_{lcl}) and the time for each local checkpoint (see Table 17). The interval between each local checkpoint is dependent on the number of failures from which application can recover from local NVM. More formally,

$N_{lcl} = T_{compute}/MTBF_{lcl}$

$T_{lcl} = N_{lcl} * t_{lcl}$

$t_{lcl} = chkpt.datasize/NVMBW_{core}$

The time per each local checkpoint is dependent on the per-process checkpoint size and the effective NVM bandwidth. As the number of cores per node increases, the effective bandwidth per-core can substantially reduce. *Hence to improve the performance of local checkpoint, we need methods that reduce the impact of this limited bandwidth, thereby improving the local checkpoint overhead and performance.*

Reducing remote checkpoint overhead. The remote checkpoint overhead for the synchronous coordinated checkpoint depends primarily on the effective interconnect/network bandwidth (i.e., InfiniBand in our case) to move data to the remote node.

$T_{rmt} = N_{rmt} * t_{rmt}$; N_{rmt}- no. of remote checkpoints

$t_{rmt} = chkpt.datasize/datamovementcost(G/sec)$

However, for asynchronous checkpoints, the remote checkpoint can be overlapped with computation, and its overhead is primarily the noise factor it imposes on the application execution. Prior studies have shown that, overlapping checkpoint data movement with communication intensive application can cause substantial overhead (e.g., 25% in [165, 29]). The communication overhead is primarily due to bandwidth contention between checkpoint data movement and application communications. Other relatively minor noise factors include CPU and memory. Hence, the equation of remote checkpoint can be rewritten as,

$o_{rmt}(sec) = \alpha_{comm} + \alpha_{others}$

where o_{rmt} represents the remote checkpoint overhead, α_{comm} the application communication overheads due to asynchronous remote checkpoint, and α_{others} other overheads like memory and CPU.

Therefore, we require mechanisms which leverage NVM to reduce the bandwidth contention between application and remote asynchronous checkpoint without affecting the remote checkpoint interval.

Restart. With increasing failures, the restart and recomputation time of an application directly impact the performance of an application. The recomputation and restart time depends on the local node and remote node failure rates. Using a node local NVM that can survive software failures (including system reboot) can substantially improve restart time performance.

$T_{restart} + T_{recov} = T_{lclrstart} + T_{lclrecomp} + T_{rmtrstart} + T_{rmtrecomp}$

The total time spent on restart depends on the local and remote restart and recovery time

Local failures: We assume that on average a computation fails half way between compute interval (I), as a result after fetching a checkpoint from local node (R_{lcl}) and I/2 of the computation must be re-executed

$F_{lcl} = T_{total}/MTBF_{lcl}$

$T_{lclrstart} + T_{lclrecomp} = F_{lcl}(R_{lcl} + I/2)$

where F_{lcl} denotes the no.of local recoverable soft failures

Remote failures: Since, we use node local NVM, our model assumes higher local NVM recovery compared to total remote NVM recovery. As a result, the interval between each remote checkpoint can contain several local checkpoints denoted by K. We assume that on average half of the segment (see Figure 61), providing one or more compute and local checkpoints is completed when a non-recoverable node failure occurs.

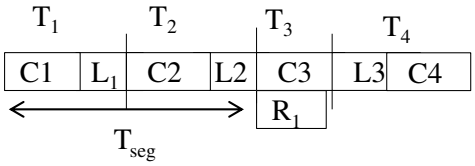
$F_{rmt} = T_{total}/MTBF_{rmt}$; F_{rmt} - denotes the number of hard errors

K = number of local checkpoints in a remote checkpoint interval

$T_{rmtrstart} = F_{rmt} * R_{rmt}$; total time spent on remote fetch

$T_{rmtrecomp} = F_{rmt} * K(I + t_{lcl})/2$; wasted computation which includes compute and local checkpoint

In this dissertation, we do not focus on improving the restart time performance, but we assume the local and remote restart time proportional to local and remote checkpoint time.



T_i - time for compute + local checkpoint.
T_{seg}- specifies remote checkpoint interval
C_i - compute time
L_i-denotes local checkpoint
R_i specifies remote checkpoint time

Figure 61: Multilevel checkpoint timing diagram.

Similar assumptions have been made in prior work [62].

7.1.3 NVM-checkpoint key mechanisms

We next present the rationale for the key mechanisms and design decisions of NVM-checkpoints, including the use of NVM as 'memory' to leverage NVM advantages like byte addressability, and hardware support for VM management, protection and caching. Additional system methods like shadow buffers and pre-copy, deal with NVM access latency and bandwidth limitations.

NVM-checkpoints uses NVM for both local and remote checkpoint, where remote nodes can be peer compute or dedicated I/O nodes. Unlike prior work [165, 63] which is completely dependent on DRAM memory either for the local or remote checkpoint, we reduce the DRAM memory usage by using NVM for local, remote, as well as intermediate buffers for data movement. With memory size predicted to be a bottleneck for exascale machines, using NVM can be highly beneficial. As discussed in Section 7.1.2 above, remote node checkpoints are less frequent compared to local checkpoints and the selection of the interval depends on hard error failure probability, i.e., failures that make a compute node unusable and no data can be recovered.

Overcoming slow NVM write using shadow buffering. Table 16 shows the five-year projection for PCM load/store latency and bandwidth [18]. The NVM write (store) operation is almost 10x slower compared to DRAM, while read latencies are comparable. Considering the high write latencies, exposing NVM directly to applications (e.g., as heap

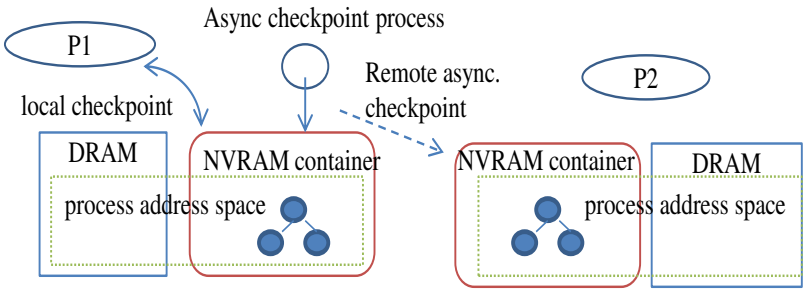


Figure 62: NVM-checkpoint architecture.

memory used during the computation) will depend on the load-store ratio of the workload. There are several research proposals to overcome slow writes by hiding NVM behind large intermediate cache. Some techniques propose memory interleaving techniques [109]. But considering the scale of HPC applications, exposing NVMs directly to the application may result in a severe application slowdown. Recent analysis by Li et al. [99] for HPC applications shows that application with higher write ratio can be slowed by 25% when directly using NVM. For frequent checkpointing, writing data directly to NVM will be inefficient.

Therefore, to overcome the previously discussed limitations, we use a *shadow buffering* mechanism shown in Figure 63. The shadow buffering method involves explicitly handling heap allocation for all checkpoint data structures (heap variables), and creating a DRAM and a shadow NVM memory chunk for checkpoint variables. Applications directly use a DRAM memory chunk pointer, and during a local checkpoint, all checkpoint data structures are moved to NVM chunks. The memory chunks can be of any granularity. When reading back the data, the application can directly access write protected NVM, and an attempt to modify the data would move the data back to DRAM for further writes. Shadow buffering is an effective technique when write sizes are large (in MBs), as in case of HPC checkpointing [36].

Dealing with limited NVM bandwidth using pre-copy. Future exascale systems are predicted to have at least 128 cores, and with increasing core count, the sustainable memory bandwidth per-core (including NVM bandwidth) can reduce significantly. Figure 64 shows the effect of increased core count on effective per-core DRAM memory copy bandwidth

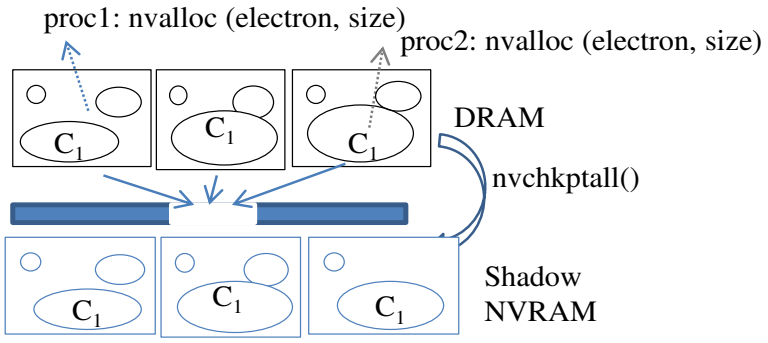


Figure 63: Shadow buffering for NVM.

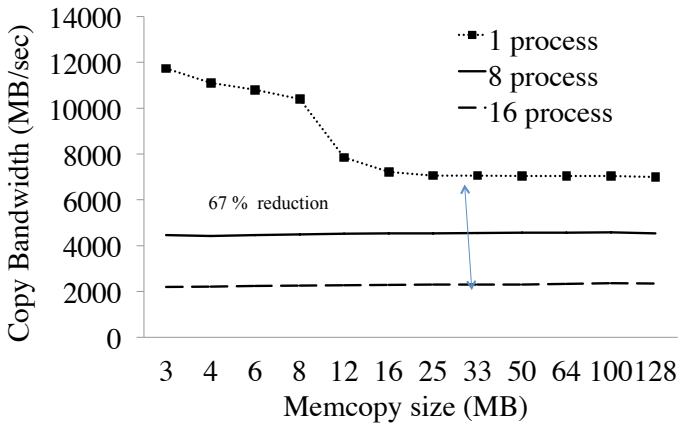


Figure 64: Memcopy bandwidth for parallel process.

using the LANL parallel memory copy benchmark [11]. As seen from the figure, with increasing core count, the per-core bandwidth reduces by 67% even for a data size of 33 MB. For NVMs with 2 GB peak device bandwidth and DDR based interface, the effective per-core bandwidth can be as low as 400 MB/Sec in a 12 core/node configuration. Hence, using a shadow buffering technique alone may not be sufficient for an application with gigabytes of per node checkpoint data to overcome the NVM limitations. Dong et al. [63] were first to discuss the DRAM-NVM memory contention issue for transparent checkpoints and proposed a thread-level serialization approach of using a dedicated checkpoint core for background copy. However, using serialization for checkpointing would incur high locking overheads and lead to slower checkpoints when the total checkpoint data size is less than the effective per-core bandwidth.

To reduce the NVM bandwidth contention for a local checkpoint, we propose a *chunk-level pre-copy* mechanism. The idea is to move the application allocated chunks (data structures) asynchronously to NVM before a coordinated checkpoint is started by partially overlapping computation with checkpoint. Three important questions need to be addressed for verifying the feasibility of this approach: 1) When to start chunk-level pre-copy to NVM?, 2) What happens if chunks are modified after pre-copy?, 3) What happens if a checkpoint fails before completion?

To answer these questions, we adopt virtual machine migration techniques where memory pages are moved to a destination incrementally even before the virtual machine is halted by using page protection methods to capture changes to a page and move them. The incremental movement reduces the migration time and peak I/O (network) bandwidth usage compared to moving all data at once. To deal with a crash and recovery, we maintain two chunk versions – a most recent completed checkpoint and a one that's currently in progress, i.e., data being modified under pre-copy. By using NVM as memory as opposed to I/O device, we leverage hardware-supported protection mechanisms, and therefore can adopt this type of incremental technique.

Chunk-based pre-copy (CPC). A key aspect of our incremental approach is to move data in memory chunks as opposed to a memory page. Applications allocate data structures to heap which include variables that need to be checkpointed. The checkpoint variables – corresponding to chunks – are allocated using NVM specific interfaces and may vary in size from a byte to hundreds of megabytes. When all compute cores in a node write to NVM during a coordinated local checkpoint, the bandwidth becomes a bottleneck and such limitation increases with increase in the number of processors. Incremental data movement is a widely accepted mechanism for reducing such bottlenecks, for instance, most of the transparent checkpoint mechanisms 'pre-copy' process pages incrementally with write protection enabled and capture further modifications by handling protection faults. One prominent issue with the incremental technique is that, when most of the data frequently changes, protection fault overheads can be high, negating the benefits of page pre-copy. For instance, handling a page protection fault can take 6-12 μsec, and 3 sec. for 1 GB

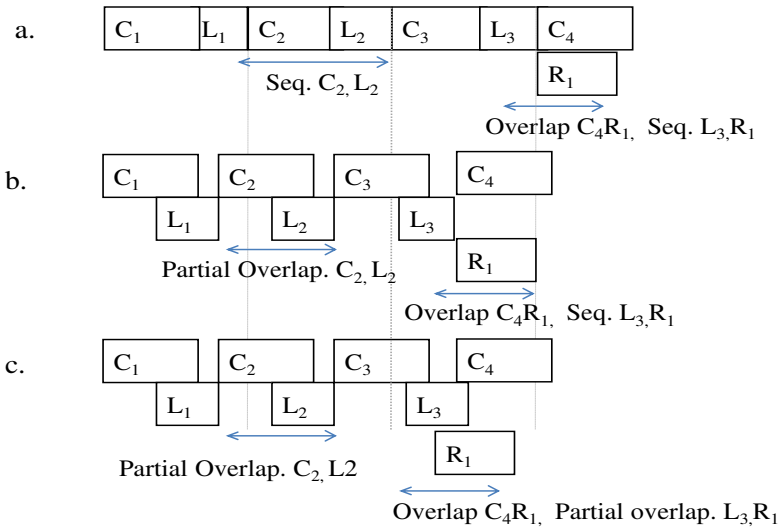


Figure 65: Checkpoint pre-copy timing diagram. C, L, R denotes compute, local and remote checkpoint respectively. Figure a. shows sequential local, non-blocking remote checkpoint. Figure b. shows overlapping compute and local checkpoint. Figure c. shows pre-copy with overlapping compute, local and remote checkpoints.

of data. Specifically, for application-initiated checkpoints in HPC applications, since most checkpoint data structures fully change, using page-level pre-copy will not be beneficial.

To avoid such costly operation, we use a chunk-level protection, i.e., all pages of a chunk allocated using the NVM interface are pre-copied in the background, even before a coordinated local checkpoint, and all chunk pages are write-protected. After the pre-copy step, when a page belonging to a chunk gets modified, the entire chunk is marked dirty (i.e., by removing protection for all pages) and pre-copied again in the background. During a coordinated local checkpoint, only the remaining dirty chunks are copied, thereby reducing the total data movement size to NVM and reducing the peak bandwidth utilization.

Figure 65 shows the timing diagram for both cases, the coordinated local checkpoint without pre-copy on the top graph, and NVM-checkpoint pre-copy scheme where local checkpoint is partially overlapped with compute phase. One issue with the pre-copy approach is that, for applications that randomly modify chunks, there may be significant application overhead due to repeated pre-copy of data from DRAM to NVM. As a side effect, pre-copy increases the protection fault overhead and the amount of data movement

across the DRAM-NVM boundary. To address this issue, we derive two variations of the basic pre-copy technique.

Delayed chunk pre-copy (DCPC). Starting local checkpoints at the beginning of a compute iteration is unnecessary, as there are several data structures that repeatedly get modified before a checkpoint, and intuitively it is sufficient to start the pre-copy sometime just before the checkpoint step. As a first step, to determine the checkpoint interval, our method waits for the first checkpoint step to complete and finds the approximate checkpoint interval (checkpoint time - compute start time) along with per-core checkpoint data size. Next, we determine a pre-copy starting time called pre-copy threshold using the below equation. We continuously adapt the pre-copy threshold to deal with application changes across iterations.

$T_c(sec) = D/NVMBW_{core}$,

$T_p(sec) = I - T_c$,

where T_c is checkpoint time, T_p is pre-copy threshold time, D is checkpoint data size (MB), I is the checkpoint interval, and $NVMBW_{core}$ is the effective NVM bandwidth per-core.

Delayed pre-copy with prediction (DCPCP): While pre-copy threshold reduces the need for repeated pre-copy of chunks, in some cases chunks can be modified until the completion of a compute phase (i.e., even after the pre-copy threshold). For instance, in a molecular dynamics application (Lammps), we observe that a three dimensional result array with relative molecular positions in a lattice gets modified until the end of a compute iteration. We call these variables *hot chunks*. Applying a pre-copy or delayed pre-copy can increase the work done (i.e., repeated copying) and as a side effect, increases the dirt tracking and data movement cost. The pre-copy overhead for such hot chunks should be reduced, or no pre-copy should be applied. To enable this, we use a simple prediction table mechanism which captures the frequency of chunk modification by maintaining a counter for each chunk and a state machine representing the modification order.

Figure 66 shows the chunk modification state machine for Lammps and for the brevity, only three out of the 31 chunks are shown. During the initial learning phase (first checkpoint), chunks are tracked for changes and the prediction counter is updated. For subsequent iterations, when the processor issues a write fault, the chunk corresponding to the faulting

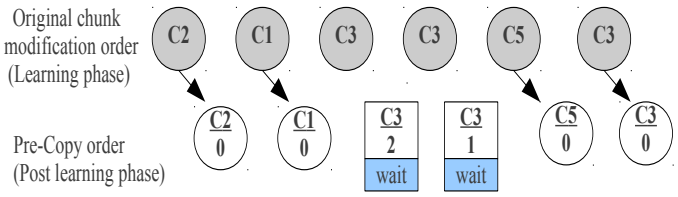


Figure 66: PreCopy with prediction.

address is marked dirty, but not copied to NVM until the modification count is equal to or greater than the value in the prediction table. For instance, the chunk C3 is modified three times during the initial run. In subsequent runs, it is not copied until the counter becomes 0. These delay-based mechanisms are optimizations and do not affect the data consistency. Hence, if the prediction fails, the data would be copied during the coordinated checkpoint step. Our analysis on three large-scale HPC applications, GTC, Lammps, and CM1 code show a relatively constant modification order. This is not surprising, because iterations are repeated without change in initial input. Our evaluation section shows the benefits and implications of the pre-copy schemes.

Using pre-copy for remote checkpoint. Remote checkpoints are necessary for tolerating node and rack failures. Remote checkpoints can be done to other compute node, or a dedicated I/O node or to the PFS. Zheng et al. [165] recently showed that, just by adding one more level of checkpointing to a buddy compute node in a different rack, the probability of unrecoverable failure can be as low as 0.000977% for an MTBF of 20 years per node, 5000 nodes, checkpoint interval of 6 minutes and 1200 hours of application time. While the checkpoint time is directly proportional to the available bandwidth between the compute and the remote destination node, remote checkpoints can be overlapped with the next phase of the computation, without requiring the application to block.

Overlapping remote checkpoints with computation result in a noise making the application slower. The noise in this context can be categorized into 1) communication noise caused by the interconnect contention between a communication intensive application and asynchronous checkpoint data movement, 2) computation noise from an asynchronous checkpoint process interfering with the application (stealing CPU cycles), and finally, 3) application

blocking overhead when there is insufficient buffer space for holding data until remote checkpoint. For reducing communication noise, prior work [165] proposed scheduling algorithms to serialize communication and remote checkpoint, or delaying the remote checkpoint until application communication is complete which can increase the remote checkpoint time. The computation noise can be mitigated by utilizing unused cores ('helper cores') [39], whereas, blocking can be avoided by using disk or NVM as a buffer. In this thesis, we focus on reducing the communication noise by using a pre-copy mechanism.

Chunk-based remote checkpoint. Our major goals of the remote checkpoint are to 1) reduce peak resource usage and 2) make remote checkpoint faster without affecting application performance. Based on these two goals, we propose a remote memory model where applications can allocate, access and copy NVM buffers to local as well as remote destination nodes. The novelty of our approach is that the applications do not have to wait for a local checkpoint to complete before a remote checkpoint starts. Instead, chunks are 'pre-copied' incrementally to remote nodes as they are modified, similar to local checkpoint pre-copy discussed earlier. By overlapping remote checkpoint with local checkpoint and compute phase (see Figure 65), the available time window for remote data transfer is increased, thereby reducing the necessary blocking time for remote checkpoints. Also, moving data in the granularity of chunks instead of moving all checkpoint data at once reduces peak interconnect bandwidth usage for large checkpoint data. Finally, maintaining separate versions for the most recent completed checkpoint vs. the one currently in progress helps protect data consistency in the event of crashes.

To fully benefit from the memory nature of NVM devices, and with the assumption of future support for DMA operations between the interconnect and NVM, the NVM-checkpoint remote pre-copy operations are designed to leverage RDMA capabilities of high-performance fabrics. Specifically, our implementation extends the widely used aggregate remote memory copy (ARMCI) library.

A helper multithreaded, asynchronous process on each physical node is responsible for remote checkpoints. The helper thread utilizes our shared NVM (similar to shared memory) support to access local checkpoint chunks and pre-copies by tracking dirty chunks in NVM.

Table 18: NVM checkpoint interfaces.

Method	Description
$genid(varname)$	generate id from varname
$nvmalloc(id, size, pflg)$	allocate NVM memory. pflg -whether data should be persistent
$nv2dalloc(..dim1, dim2..)$	2D Fortran allocation wrapper
$nvattach(id, src, size)$	create shadow NVM copy for exisitng DRAM memory
$nvrealloc(id, src, size)$	grow memory
$nvchkptall()$	checkpoint all persistent chunks
$nvchkptid(id)$	checkpoint specific chunks/variables

As a design optimization to decouple the helper process from applications, we introduce additional APIs and system calls to query our library about a process' NVM structure. Section 7.1.4 discusses this in greater detail. Further, to reduce wasteful bandwidth usage, we use the delayed pre-copy with prediction (DCPCP) mechanism for remote checkpointing. The delay time before a remote pre-copy is dependent on the remote checkpoint interval.

7.1.4 Implementation

We next discuss the implementation of the system- and application-level NVM-checkpoint components.

NVM kernel. In our hybrid memory design model, the NVM virtual memory support is enabled by an NVM kernel manager which extends the Linux memory (DRAM) manager. We use the NVM kernel manager implementation discussed in Chapter 3 and extend it with checkpointing-related support, including for pre-copy and shadow buffering.

The NVM memory manager is responsible for NVM-based paging and in-kernel process-level persistent data structures. To emulate future NVMs like PCM, the kernel manager reserves a fixed physical address range in DRAM during OS boot and manages these pages along with persistence support. Further, a set of system calls is exposed to allocate, extend or delete NVM pages. Allocations are done using the *nvmmap()* system call similar to the *brk()* or *mmap()* system call for heap. We maintain a metadata structure for each process that keeps track of all NVM pages used by a process. During applications restart, the information in the metadata structure along with the system call arguments are used

to load the persistent pages to the process address space. Also, since NVM utilizes the processor cache to reduce the NVM read/write latency, all data in the cache is flushed to NVM before data is marked consistent, using the Linux cache flush kernel method. All data structures and the application variables that need to be checkpointed are allocated using NVM user-level interfaces (described below) and maintained on a per-process basis.

For supporting pre-copy, the asynchronous checkpoint process on each node should have access to all of the checkpoint process metadata from which the checkpoint data regions can be identified. To enable this, we provide a system interface which loads the entire metadata structure to the process address space. The metadata structure is protected by locks to avoid conflicts between the application and remote checkpoint process. Next, since two versions of checkpoint data (i.e., a committed version and current uncommitted version) can reside as a local checkpoint, we add multi-version support for our internal data structures. As an optimization, while our DRAM to NVM local checkpoints uses page protection techniques to identify dirty chunks, in the case of remote checkpoints, to avoid frequent protection faults, we add a 'nvdirty' bit for each NVM page supported by a system call to identify dirty NVM pages of a chunk.

NVM user library. Our user library provides NVM allocation, checkpoint and restart interfaces for applications. Table 18 list few important interfaces. The user library consists of a) an NVM allocation component, b) shadow buffering and checkpoint component and c) a restart component.

a) *NVM allocation component.* All data structures/variables that need to be checkpointed are allocated using the *nvmalloc()* interface and referred to as chunks. Each chunk has metadata structure and a corresponding NVM data region. The allocation component extends the highly scalable Jemalloc allocator to manage allocations to NVM using the *nvmmap()* interface and maintains a user-level persistent data structures for each such allocation. During chunk allocation, the allocator creates a DRAM chunk and a corresponding NVM chunk and returns the DRAM pointer to the application. Every chunk is identified by a unique identifier supplied by the application. All chunk metadata of a process is stored in a separate per process metadata region, not directly accessible by the application.

b) *NVM checkpoint component:* The checkpoint component is responsible for shadow buffering (i.e., moving data from DRAM to NVM), tracking chunk-level modification for local and remote chunk-level pre-copy, ensuring the consistency of checkpoint and also managing checkpoint versioning. When an application invokes the *nvchkptall()* interface, all checkpoint variables are moved from DRAM to NVM. Our library uses its metadata structure to identify all chunks and moves them to NVM. All chunks are marked as committed after the library ensures that data is flushed to NVM. For enabling pre-copy, each chunk structure has two dirty bit flags, one for a local checkpoint and another for a remote checkpoint. These dirty bits are used to identify chunks that need to be pre-copied. To deal with failures, we maintain two versions for each chunk, a previously committed chunk, and chunk currently being written. When a checkpoint fails, the library reverts to the committed version for recovery. When local NVM space is a constraint, only one version is maintained locally and on checkpoint failure, the application retrieves the chunk data from remote nodes.

c) *Restart component:* During restart, the applications use the same *nvmalloc()* interface with a unique identifier and the persistent flag as the argument (see Table 18) to read back data from NVM. The persistent flag specifies a condition whether to read previous checkpoint data if data exists. An optional feature is the checksum capability, where after every checkpoint, a chunk data checksum is calculated and stored along with the chunk metadata. On a restart, the checksum is recalculated and verified against the metadata. The restart component first checks if the checkpoint data is available/consistent and if not, fetches the data from the remote peer node. Our current restart mechanism is simplistic, and our future work will consider better optimizations.

Application Modifications. Our current design requires application-level changes for adapting to NVM based checkpointing. Specifically, applications need to use the NVM allocation interface for variables/data structures that need to be checkpointed. Currently, our library provides FORTRAN and C/C++ interfaces, which are being hardened as we gain experience with more diverse applications. For example, when adding NVM support to the Lammps application, we noticed that it relies extensively on its custom user-level memory management. Modification of these types of applications can be tedious. Similarly,

in some applications, the checkpoint size cannot be statically determined. Such applications can use our *nvattach()* interface for lazy creation of a checkpoint chunk by attaching the DRAM pointer with a shadow NVM pointer and subsequently use the *nvdelete()* method to let the NVM library remove references to the chunk.

7.1.5 Evaluation

The overall goal of NVM-checkpointing is to leverage future NVM present in next-generation exascale machines in reducing checkpointing overheads without compromising application reliability. Our experimental analysis uses the performance model described in Section 7.1.2. We categorize the checkpointing overhead in three components: (1) local checkpoint time, (2) remote checkpoint overhead, and (3) overall resource utilization. Our experiments focus on the key issues affecting the performance of each of these three components and demonstrate the ability of NVM-checkpoint to reduce the checkpointing impact on application performance. Specifically, we evaluate the following:

- Reducing local checkpoint time: We evaluate the effectiveness of NVM-checkpointing to deal with NVM bandwidth limitations and to reduce local checkpoint time.
- Reducing remote checkpoint time: We measure the ability of our remote checkpoint design in improving application efficiency.
- Resource utilization: For both local and remote NVM-checkpointing we evaluate the overall system resource utilization (e.g., interconnect bandwidth, CPU, and memory) of our methods.

Methodology. All experiments are conducted using an eight node cluster, with each node consisting of 12-2.8 GHz Intel Xeon cores, 48 GB memory, and 40Gbps InfiniBand. We use the mvapich2 MPI library for all applications, with one process per-core. For NVM emulation, we partition half of the DRAM for NVM-related allocations managed by our NVM component. Persistence across application session is provided by locking the DRAM pages from being swapped out or freed. To emulate different NVM bandwidth, we introduce data copy delays derived using the LANL memory copy benchmark similar

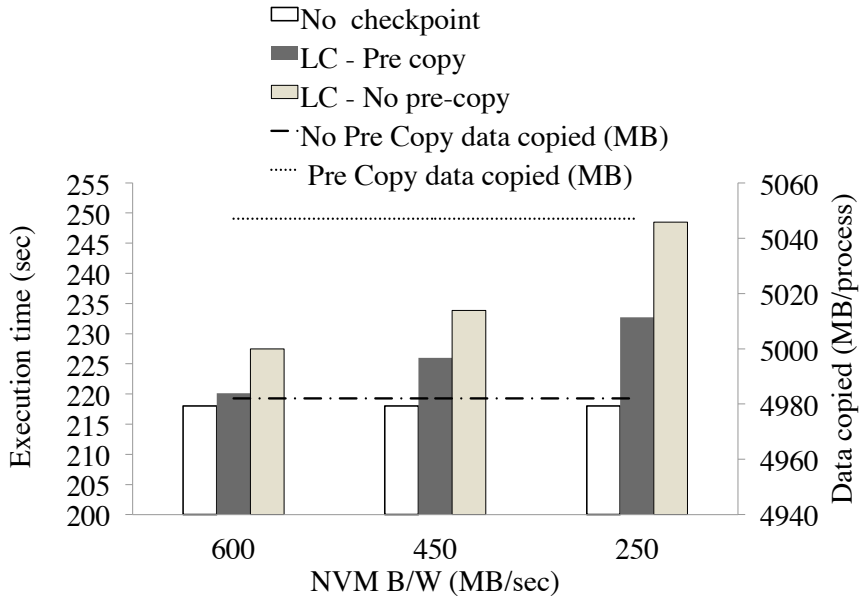


Figure 67: Lammps - Local checkpoint Pre-Copy vs. no Pre-Copy.

to [146]. We assume the NVM device bandwidth as 2 GB/sec [18] and vary the effective per-core bandwidth.

Applications. To understand the impact of NVM-checkpoint on real-world HPC applications and their checkpointing requirements, we study the checkpointing behavior of three large-scale widely used HPC applications: 1. Gyrokinetic Toroidal Code (GTC) is a 3-Dimensional Particle-In-Cell code used for studying microturbulence in magnetic confinement fusion from first principles plasma theory. The checkpoint data primarily have 2D arrays representing electrons and ions. The application is highly scalable, and each core can output two million particles roughly every 120 seconds resulting in 260GB of checkpoint data. 2. Lammps is a well-known particle dynamics code that supports a wide variety of simulation techniques applicable to biology, chemistry, and material sciences and we use Lammps benchmarks with configurations similar to [10] and use Rhodo suite for our analysis. 3. CM1 is a three-dimensional, non-hydrostatic, non-linear, time-dependent numerical model suitable for idealized studies of atmospheric phenomena. For our experiments, we examine the 3D hurricane simulation. The CM1 code is an open source Fortran code and uses a similar checkpointing mechanism to GTC application.

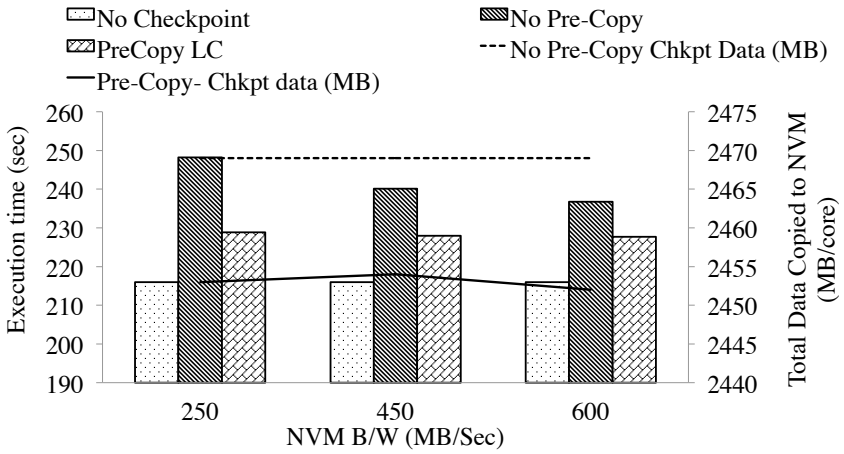


Figure 68: GTC - Local checkpoint Pre-Copy vs. no Pre-Copy.

NVM local checkpointing. Figures 67 and 68 compare the local checkpoint performance of a pre-copy checkpoint against a no pre-copy case. For no pre-copy, local checkpoints and compute step are not overlapped, and local checkpoint is started only after a compute cycle is finished. The x-axis shows different NVM bandwidth/core estimates, the left y-axis shows the application execution time and the right y-axis – the total data copied to NVM for local checkpoints. For Lammps, the experiments are done with 48 MPI processes using the RhodoSpin benchmark, and checkpoint size/ process is ˜ 410 MB. We choose the RhodoSpin benchmark as it checkpoints relatively higher number of chunks that are modified across different application stages. As shown in Figure 67, even with decreasing NVM parallel bandwidth, pre-copy checkpoint adds only 6.5% overhead to application execution time, compared to the 15% in the no pre-copy case. Overall, the pre-copy method improves performance by 15% compared to the RAMDisk approach. Chunks that are modified just before the checkpoint step cannot benefit from a pre-copy prediction mechanism and result in additional work. As it can be seen, the total data copied by pre-copy is slightly higher (3%). For GTC we use the same processor count and checkpoint data size. The application shows similar benefits from using the pre-copy approach (see Figure 68). An interesting point to note is the reduction in checkpoint size for the pre-copy case. For GTC, we observe that few large chunks (variables) are modified only once (during application initiation). Similar observations can be made for Lammps too, though the aggregate data

Table 19: Chunk size distribution range in percentage.

Application	500K-1MB	10-20MB	50-100MB	above 100MB
CM1	40	0	54	4
GTC	45	9	0	45
LAMMPS	15	0	20	25

size of these variables is less significant. By leveraging the memory protection mechanisms in NVM-checkpoint, we can efficiently track chunk-level modifications, and avoid repeating checkpoint for unmodified chunks without more heavy-weight diff computations. The combined use of pre-copy with the reduction of total checkpointing data size improves the local checkpoint performance of GTC by 10%, compared to the no pre-copy case.

The CM1 application (not shown for brevity) shows less than 5% benefits from the pre-copy approach. To understand the reason for such variation across application, we analyze the chunk size distribution. Table 19 shows the checkpoint chunk size distribution for the three applications. We examine the impact of chunk sizes on pre-copy performance for a fixed checkpoint size (400MB) and checkpoint frequency. We categorize chunk sizes into different ranges varying from 500KB to over 100MB. In the case of CM1, about 40% of the chunks are less than 500K and around 50% of chunks less than 50 MB. In the case of GTC and Lammps about 50% and 30%, respectively, of the chunks are larger than 100MB. The NVM bandwidth limitation, which pre-copy attempts to alleviate, causes more significant levels of contention for large chunk sizes. This behaviour explains the observed benefits of pre-copy for Lammps and GTC, and the more modest impact of the approach on CM1, where less than 5% of the chunk exceed 100MB.

NVM remote checkpointing. We next analyze the impact of the NVM-checkpoint remote checkpointing mechanism on the application efficiency. We define efficiency as the ratio of ideal application run time to actual run time. The ideal run time represents a case no fault occurs, and the application does not checkpoint, whereas the actual time represents the application run time with local and remote checkpoints. Recent studies have shown a large variation in estimating the MTBF for exascale systems. We use the failure rates (λ)

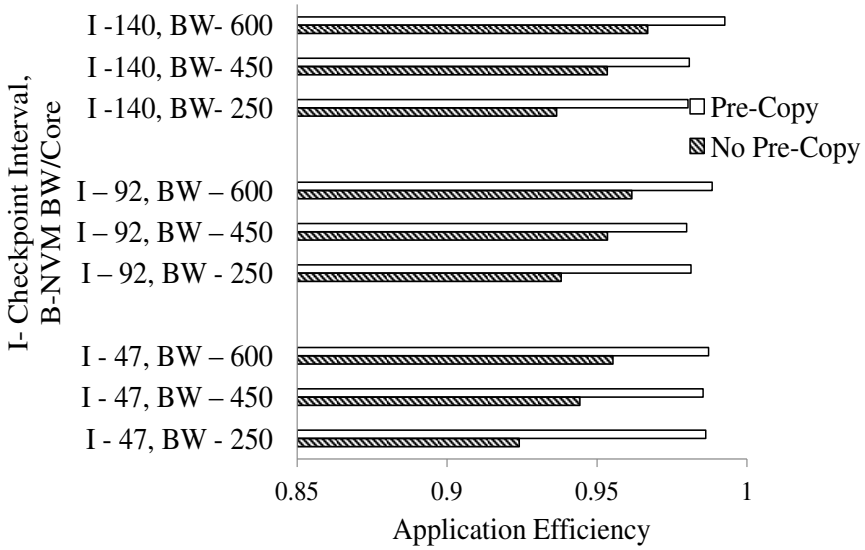


Figure 69: GTC - Remote checkpoint efficiency.

and optimal checkpoint interval (30 to 100sec) estimated by X. Dong et al. [62], and vary the ratio of permanent (λ_{remote}) vs. transient (λ_{local}) failures. Considering our experiments scale, we set the local checkpoint frequency to 40 secs. Figure 69 compares the remote checkpoint overheads of an asynchronous pre-copy and asynchronous no pre-copy for GTC. In our experiments, the average checkpoint data per-core is around 433MB and 4.7GB per node. We vary the remote checkpoint frequency from 47 to 180 seconds. The horizontal axis shows the application efficiency compared to the baseline case without any checkpointing. From the figure, we observe that even at reduced levels of NVM bandwidth, remote pre-copy checkpointing delivers significant improvements in achieving application efficiency, compared to the no pre-copy case. With the increase in available NVM bandwidth, and at increased checkpointing intervals, NVM-checkpoint can achieve application efficiency by 0.98. We observe similar trends for the other applications studied in this chapter. On average, across the three applications, pre-copy based remote checkpointing adds 6.2% to the application runtime, compared to 10.6% of the no pre-copy approach, representing a reduction of nearly 40%. At increasing system scales, this can translate to substantial gains.

Several studies in the past have reported substantial communication interference (close to 22% as reported by F. Zhen et al. [165]) when overlapping the asynchronous checkpoint

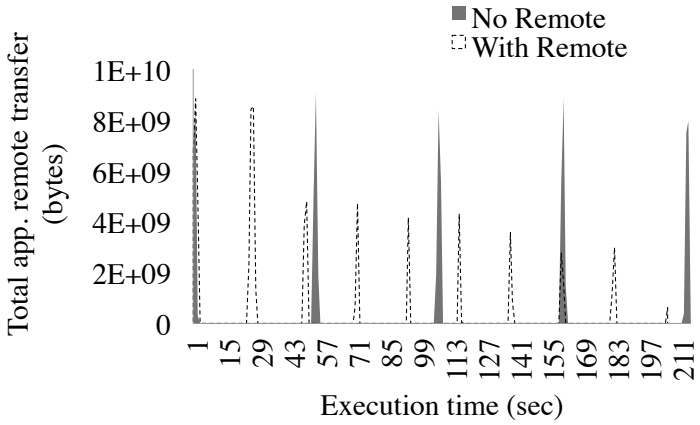


Figure 70: Lammps - Peak interconnect bandwidth usage.

with computation. Such interference is primarily due to higher peak bandwidth utilization of a remote asynchronous process, causing interconnect contention. In the case of the pre-copy approach, by increasing the time window of the checkpoint, and accessing/moving checkpoint data in chunk granularity, the contention is more likely to be avoided, at the cost of a potential increase in total checkpointing data volume. Figure 70 compares the peak interconnect usage of the no pre-copy and pre-copy-based remote checkpointing. The y-axis in the figure shows the total checkpoint data transferred across the entire application, and the x-axis shows a timeline of the application execution. Clearly, no pre-copy requires moving all data at once, which substantially increases the peak interconnect usage. In the case of the pre-copy based approach, the peak resource usage is almost half the 'no pre-copy' case –, i.e., by avoiding the need to move all data at once pre-copy substantially reduces the interconnect contention. The high peak resource usage in the initial application stages of the pre-copy approach is due to the learning phase when our library learns about approximate remote checkpoint interval and checkpoint data transfer time. Thus, the figure also demonstrates the utility of the 'delay-based' optimizations of the pre-copy approach. Finally, the pre-copy methods require the use of asynchronous pre-copy threads and additional CPU resources. We find that the average CPU utilization of the dedicated checkpointing core, executing the pre-copying operations, doubles (see Table 20), however, it remains at relatively low levels when compared to the node-wide CPU utilization – at

Table 20: Checkpoint helper core average CPU utilization.

Data/core(MB)	No Pre-Copy- Util%	Pre-Copy- Util%
370	12.85	24.48
472	13.40	25.12
588	14.82	28.31

˜2.5%.

7.1.6 Summary of NVM checkpointing

We first analyzed the role of future NVMs like PCM and Memristors in reducing the checkpoint cost, critical for application speedup. We adopt the state-of-art "multilevel" checkpointing approach and discuss the importance of a hybrid, i.e., node local and remote checkpoint approach. To overcome high write latency cost and bandwidth limitations of NVM, we treat NVM as a virtual memory and propose three novel pre-copy techniques supported by a shadow buffering mechanism to overcome such hardware limitations. Further, we extend our NVM system design by enabling remote memory operations supported by our pre-copy schemes and demonstrate a reduction in non-blocking remote checkpoint overhead and peak interconnect utilization.

7.2 Heterogeneous memory in browsers

Browsers have become an indispensable computing platform for client devices, ranging from cell phones, laptops, tablets, and desktops, not just for web browsing, but rather a complete computing framework. Most browsers today support direct access to the underlying hardware and accelerators. Examples include HTML5 support for multi cores, WebGL APIs for GPUs. However, with growing computing needs, large data access and storage needs have also increased as fetching data from the network is time-consuming. Widely used runtimes such as HTML5, JavaScripts, and Google Native Client (NaCl), have started supporting direct I/O access for web applications. The local storage interface for browsers [5] exists in multiple forms like (1) simple key-value store, (2) JavaScript (JS) based SQLite interface, (3) synchronous and asynchronous POSIX I/O interfaces for storage of large blobs of data. All of the above methods are compatible across different browsers but are limited by JS

and dynamic compilation bottlenecks. State-of-the-art frameworks such as Google's NaCl, support richer applications written in native languages (C, C++) that are 4-5x [156] faster than JS. However, providing direct access to the web application for underlying storage resource can leave the system in a vulnerable state due to security threats. A commonly used solution is to isolate the untrusted web applications from the trusted browser framework and underlying OS by 'sandboxing' [148, 144, 156]. Sandboxing enables secured access to system resources like memory, network, and storage, by intercepting applications' access to these resources (i.e., system call interception). However, as a side effect, Sandboxing, by adding one more level of indirection, significantly impacts I/O performance. Specifically for resources with software-controlled access, the overhead of sandboxing is higher. Frequent I/O calls for sandboxed application can substantially increase I/O latency and reduce throughput irrespective of the underlying storage device used. With traditional block-based storage interface, data serialization and deserialization in a sandboxed environment further degrade application runtime.

Hence, a fundamental principle to reduce sandboxing cost is reducing the software intercepted resource access without compromising the protection features of sandboxing. We use this principle, by using NVM as a virtual memory-based heap as opposed to a block-based device, and exploit the hardware controlled virtual memory-based isolation between applications. By using a VM-based interface, coupled with features like memory page protection, each web application is restricted/ isolated to access its state in a restricted boundary. While using NVM as a heap requires application-level changes, it avoids a substantial number of sandboxing interceptions (for example, intercepting every read/write call), and hence, reduces the overall resource access latency critical for end-user devices. We realize the benefit of our proposal in the Google Chrome-based Native Client (NaCl) framework. With NaCl, applications run as a browser extension across client devices supporting major OSs, like (Windows, Linux, Mac, ChromeOS). We refer to the NaCl framework as state-of-the-art which is 4x faster than their HTML5 JS counterparts. NaCl applications are developed in C, C++ experience less than 5% overhead relative their native alternatives.

I/O in browsers. I/O capabilities for in-browser web applications exist in many forms.

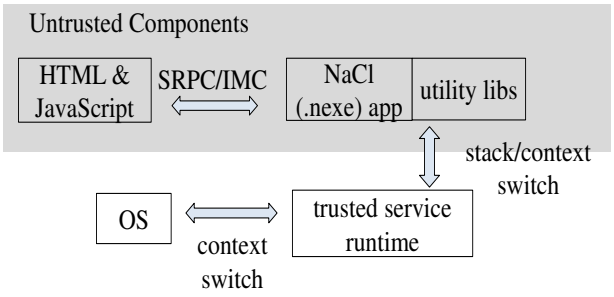


Figure 71: Multiple indirections of syscalls within a sandbox.

IMC refers to Inter-Module Communication and SRPC refers to Simple Remote Procedure Call

They include traditional application-transparent and explicit browser [153, 122], a simple key-value store used for storing user personalization information. More structured data that includes metadata storage of browser cache, browser game states, and others use a JavaScript (JS) based SQLite interface. Applications that require storage in blobs, for instance, downloading a compressed video file and decompressing it before playing, use synchronous and asynchronous file system interfaces. Yee et al. [156] discuss an interesting photo storage application using the NaCl framework to store and process images. *While the need for I/O in browsers has been increasing, poor I/O performance has continued to pose significant limitations to web application developers [22].*

Sandboxing. The key goal of application sandboxing is to isolate applications from code and data of other applications by restricting the access to system resources and to comply with user granted permissions. Sandboxing mechanisms vary across systems, ranging from rule-based executions to virtual machine emulation to static code profiling. In the case of higher-level languages like Java, language constructs, and runtimes provide the sandbox (Dalvik VM for Android-based systems), for systems that support native C, C++ languages (NaCl), a separate sandboxing layer enforces restrictions on instructions and system calls that applications can use. *But on the whole, while sandboxing is required to improve security when running untested and untrusted code, frequent access to system resources can affect performance. Specifically, access to storage device comes with increased access latency.*

Sandboxing in Google Native Client. We next provide a brief background on sandboxing and discuss the importance of choice of interface for improving the I/O performance. While we use the Google-based NaCl, other runtimes also have similar sandboxing cost. The NaCl-based browser application consists of a trusted and an untrusted component. The browser user interface and application libraries are untrusted components whereas the browser execution environment and the NaCl framework are trusted components. The trusted, and untrusted components have their own private address space similar to user and kernel layers of a traditional OS. A transition from untrusted to trusted region (or the reverse) requires stack switching. More details about trusted and untrusted components can be found in [156]. The NaCl framework adapts two levels of sandboxing – inner sandbox and outer sandbox [156]. The inner sandbox provides binary validation by using static analysis and restricts unsafe instructions. Because all analysis are done statically, the inner sandbox has a lower impact on application runtime. In contrast, for the outer sandboxing, untrusted applications' use of system call wrappers are intercepted by a trusted region. Similar to context switches between user and kernel-level, control transfers happen between untrusted (browser application) and trusted regions (trusted browser framework) using springboard and trampoline techniques, making a system call highly expensive compared to general applications. For browser-based I/O, NaCl uses the HTML5-compatible Pepper library and memory access by untrusted applications are restricted to a particular address range using page protection mechanism, and any region can be expanded/shrunk by registering it with the NaCl runtime. The runtime maintains a per-process (an untrusted application) address table mapping containing the address range and access permissions and registered address regions do not incur sandboxing costs, but rather leverage the hardware support for illegal access protection. This is in contrast to file system operations, where every I/O syscall needs to trap. *Frequent I/O calls by applications cause severe I/O and bandwidth impact irrespective of the underlying physical device (e.g., NVM, RAMDisk, or SSD), which makes such file system calls highly unsuitable in browser-based environments.*

To understand the importance of choice of interface, we did a simple test using NaCl. Figure 72 shows a simple benchmark demonstrating the I/O performance issue in browsers,

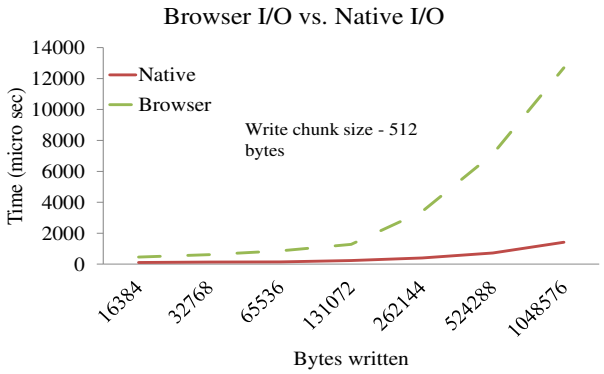


Figure 72: Sandboxing impact on IO performance.

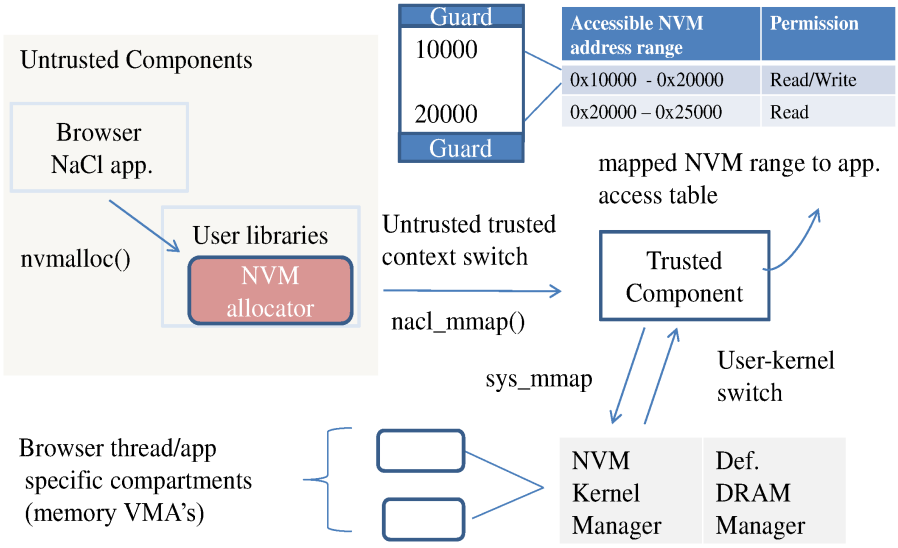


Figure 73: Design for sandboxed browser NVM support.

comparing the browser-based case with the native execution of a simple benchmark that opens a file, writes data to it in chunks of 512 bytes, and then closes it. There is a substantial increase in I/O time for the browser case, attributed to the fact that with increasing numbers of I/O calls, sandboxing overheads also increase. Use of the existing memory map system call interface when dealing with few large files can reduce sandboxing impact. However, when the number of mapped and unmapped files are high (for an example, compressing all image files in a directory by memory-mapping them), the overall user-kernel context switching can negate the memory map gains as shown in our evaluation section (see Sec. 7.2.2).

7.2.1 NVM design for sandboxed browser

I/O calls in a sandboxed environment transition from untrusted to trusted to privileged (kernel) layers. Our design reduces the multiple levels of software redirections for I/O calls, by relying on fast hardware-based page access for persistent storage. We achieve this by exploiting the byte addressability and hardware-supported page-based memory management and protection techniques for NVMs. Applications allocate persistent regions of NVM similar to a heap, and instead of file system reads and writes, perform load and store operations to the persistent regions. Main differences with prior work [147] include NVM support for and several optimizations unique to persistent browser applications in sandboxed environment and a virtual memory-based NVM kernel manager compared to the file buffer cache extensions in prior work. For OS support, we use our pVM design.

Our design consists of a user-level library to allow NaCl browser applications to allocate explicitly and access persistent data in NVM. The NaCl framework categorizes the runtime into trusted and untrusted components (see Figure 73).The trusted region implements protection; it is responsible for providing all system resource references and handles, along with system call interception. The untrusted region provides the user-level interfaces to the NaCl applications which are intercepted by the trusted runtime. Because both regions maintain separate stacks, a call from the untrusted to trusted region results in expensive stack switching. For avoiding such costs, application-level resource management can be done in the untrusted region after getting a resource reference. For instance, user-level memory allocators implemented in the untrusted region and the reference of memory addresses using the *sbrk()* or *mmap()* call can be obtained from the trusted region. To match this division of state and functionality across NaCl components, we divide our user-level NVM component across the trusted and untrusted layers of the NaCl library. We next describe the untrusted and trusted components.

Untrusted NVM allocator. The untrusted NVM component consists of the persistent memory allocator discussed in Chapter 3. The implementation is secure because a mmap-based reservation is made by the allocator using a call to the trusted region, and any illegal memory address access outside the registered range of the untrusted application would result

in an application exception. The allocator manages the persistence guarantees required for restarts.

Sandboxing specific allocator optimizations. Most optimizations revolve around reducing the frequent use of the system calls in the allocator (e.g., *sbrk(), mmap(), munmap()*), as this can negate the benefits over a POSIX I/O interface. Two key optimizations include (1) allocator memory reservation size, and (2) dividing the memory reservations among multiple threads. Regarding (1), allocators use *mmap()* or *sbrk()* to reserve a few pages of memory, try to fit in application allocations in the reserved regions, and when the reserved region is insufficient, invoke a *mmap()* call again. For applications requiring large persistent storage needs, this may result in a substantial number of system calls (and hence sandboxing overheads). Therefore, applications provide a hint to the allocator for making larger reservations, (maximum of 16 MB), with default reservations of 4MB. However, large reservations in multithreaded applications can be dangerous. Hence, we divide the application reservations into thread-level compartments discussed earlier [70].

Trusted NVM component. The trusted NVM component is a thin layer responsible for providing an indirect access to the system-level NVM interfaces like *nvmmap()* for allocating and accessing persistent regions, maintaining a per-application persistent NVM access region table with different protection levels, and handling out of bounds access protection faults. Every untrusted application registers a unique key with the trusted region for the first time, and the same the key is used across sessions. The key registration also creates a persistent access table for the application (see Figure 73). After registration, applications use untrusted allocators for persistent memory allocation, which invokes an NVM-specific *nvmmap()* call to the trusted component. The trusted component checks if the requested memory reservations are private, and adds the memory address range to the access table. Since the trusted and untrusted components have separate logical segments and stacks, after memory allocation, the trusted component converts the memory reference to the untrusted application address range.

Once the NVM address ranges are mapped into the process address space, the applications are free to access any memory address in the range and do not encounter sandboxing

costs. This provides substantial performance benefits by reducing the outer sandboxing overheads. Across application (browser application) restarts, unique keys are used as unique naming entities for reloading the applications' access table. Our current design relies on the browser application to provide a unique key which is similar to sandboxing in the Android framework [42], where each application has a unique key across sessions. Future work will focus on more application-transparent key generation.

7.2.2 Experiments

To investigate the benefits of leveraging the byte-addressability of future nonvolatile memories in improving browser application's I/O performance, we seek to understand the following. (1) Is the current storage device performance mainly responsible for the I/O slowdown in sandboxed environments like browsers? (2) What is the impact of the choice of storage interfaces on a sandboxed environment? (3) What are the benefits of treating NVMs as a nonvolatile heap as opposed to a block storage device? To answer these questions, we use the browser-based WebShootbench [27] benchmark, and two applications: Snappy data compression, and an offline content-based email classifier. We next provide details on the experimental methodology.

Evaluation Methodology. For representing an end-user devices like smartphones, we use a dual-core 1.66 GHz D510 Atom-based development kit with 2GB DDR2 DRAM, Intel 520 120GB SSD. We use the same NVM emulation discussed in Chapter 3. We observe that for most applications except for a hashtable benchmark, the cache misses were less than 1-1.5% as discussed in other work [65].

7.2.2.1 Benchmark analysis

WebShootbench [27] is an open source NaCl benchmark originally derived from the Computer Language Benchmarks Game [2] to compare the speedup of NaCl with JavaScript; we focus just on the workloads that depend on I/O. Table 21 shows the I/O vs. compute time on a vanilla Linux Atom platform. To understand the impact of the storage device, we evaluate SSD, RamDisk, and an emulated NVM.

- Fasta (FS) is a write-intensive benchmark that generates random DNA sequences

Table 21: I/O time for benchmark applications.

Benchmark	I/O time(%)
Fasta	41.207
Revcomp	49.33
kNucleotide	12.32
SpellCheck	19.89

by weighted random selection from a list of predefined sequences and writes three sequences line-by-line. The number of 'fwrites' system calls are substantial

- Revcomp (RC) reads DNA sequences line-by-line from the output generated by Fasta, and for each sequence, writes the ID, description, and the reverse-complement sequence to output. Blocking read calls dominate the I/O time of the application.

- kNucleotide (KN) reads the DNA sequence from Fasta's output line-by-line, generates k-nucleotide sequences, and each k-nucleotide is updated to a hashtable, with values representing counts of occurrences. The I/O time is less than 13% compared to the total compute time (hashing).

- Spell Check (SC) loads popular 'Wordnet' dictionary files [108] into a hashtable, generates words from an input file and identifies words that are not in the dictionary. The dictionary set contains 16 files each containing its hashtable. We use four 16MB input text files.

Figure 21 shows the time spent on I/O by each application. To understand the impact of the storage device on performance, we perform experiments with SSD, RamDisk (RD) and NVM.

Observations Figure 74 compares the use of NVM as a heap with RAMDisk and SSD performance. The applications generate/access around 64MB of I/O data. As expected, NVM as a heap shows significant performance gains (Y-axis shows runtime) in all the benchmarks with maximum gains for read-intensive 'Revcomp' (3.5x) and least gains for compute intensive kNucleotide (20%) and short running spell check application. An interesting result is that both RAMDisk and SSD perform poorly with negligible difference

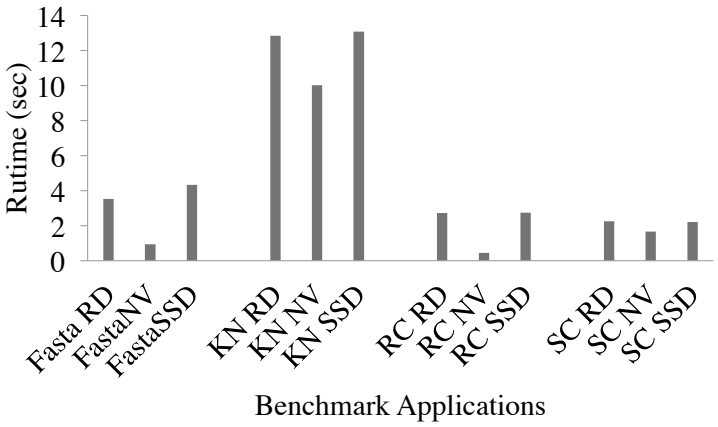


Figure 74: Benchmark performance comparison.

Table 22: NVM gains: server (Sandybridge) vs. client (Atom).

Benchmark	Server Gains(%)	Client Gains(%)
Fasta	68.85	73.51
Revcomp	73.27	83.82
kNucleotide	15.84	21.94
SpellCheck	22	26.234

between them. The results show that frequent I/O read/write calls hurts both SSD and RAMDisk, which shows the impact of sandboxing in browser I/O slowdowns. Even writes that benefits from a buffer cache, suffer substantially using the POSIX I/O interface in sandboxed machines. As expected, increasing the I/O size, resulted in a widening gap between the NVM heap and RAMDisk approaches (not shown here for brevity). We also observed that, the speedup achieved from our NVM-based design (compared to RAMDisk) on the client platform (Atom) to be higher than the server platforms (Sandybridge) as shown in Table 22. As results show, sandboxing increases total instructions executed, and reducing such actions in slower cores with our NVM design shows higher benefits. *These observations show that the choice of interface is critical in browsers, and using NVM as heaps can avoid the substantial sandboxing cost.*

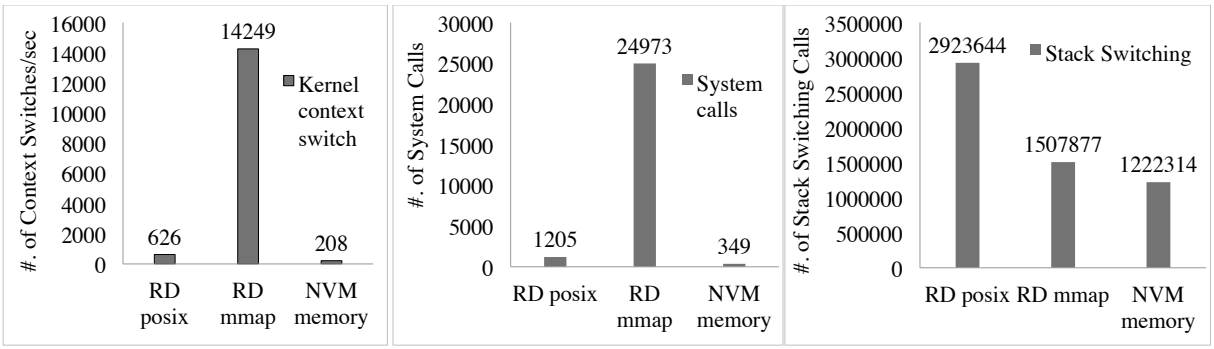


Figure 75: Snappy analysis. Figure in the left compares kernel context switches, middle figure shows number of system calls, and the figure in the right compares untrusted to trusted stack switching.

7.2.2.2 Application evaluation

We next evaluate the effectiveness of NVM as a heap for the browser I/O using two other applications, (i) a NaCl-based disk cache compression using Snappy [23], (ii) Bayesian-based offline email classifier. Using Snappy, we compare the implications of using a memory interface for NVM vs. a POSIX-based block interface or a memory map interface. With the email classifier, we analyze the serialization/deserialization benefits of NVM. For the POSIX I/O and mmap interface, we use an RAMDisk-based file system (tmpfs) to avoid the storage device noise.

Snappy Compression. Snappy is a high-performance compression/decompression library (ships with Chrome sources) and we discussed the Snappy I/O characteristics in Chapter 3. We ported it to NaCl and compress approximately 500MB of default browser cache data (3001 files), as shown useful in [156, 45]. The cache consists of binary, text, images and video files, and compression achieves 28% reduction in cache size. The compression/decompression time is well within the limits of average web page load time (∼4-7 seconds). For the mmap case, each file is mapped into memory, compressed and unmapped, whereas POSIX block refers to block I/O interface. For NVM, we use the *nvread()* method.

Figure 76 shows the runtime comparison and Figure 75 compares the context switch, system calls, and stack switching for all three interfaces. Our memory-based interfaces outperform mmap-based interface by nearly 2.5x and the POSIX I/O interface by over 3x. Analyzing the reason for the performance difference, the left of Figure 75 shows the average

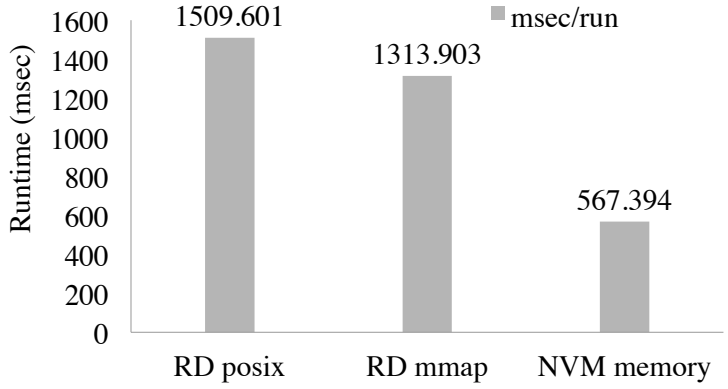


Figure 76: Snappy access interface evaluation.

user-kernel context switch counts per second. The total number of context switch when using a *mmap()* interface is significantly higher compared to to block-based *fread(), fwrite()* calls. When using the *mmap()* and *munmap()* system calls, every invocation results in a context switch as confirmed by the figure in the center which captures the overall system call invocation of the application. In the POSIX I/O interface, while *fopen(), fclose()* results in a system call, *fread()*, and *fwrite()* are library calls, which explains substantially lower context switch. But the amount of stack switching between untrusted and trusted region explains why the POSIX interface suffers compared to the mmap-based interface even though user-kernel the transition is less when compared to mmap interface. In the case of our *nvread()*, files are memory chunks (objects) with a name identifier. Hence, the read call results in mapping a 2MB region (see allocator optimizations), resulting in 2x fewer system calls compared to the POSIX I/O. Further, the application has a sequential access pattern with runtime less than a second resulting in a relatively lower cache misses.

7.2.3 Summary of browser support

In this section, we discussed the benefits of using NVM as virtual memory-based for dealing with the high sandboxing costs for I/O operations of browser-based applications. Using the Google Chrome's Native Client runtime, and several application benchmarks, we demonstrate the benefits of using PCM by leveraging its virtual memory based interface in reducing such costs.

7.3 NoSQL database redesign for heterogeneous memory

We next discuss HeteroDB – a heterogeneous memory design for NoSQL [46] databases that maintains a persistent key-value store using log-structured merge (LSM) trees [115]. An LSM-tree is a persistent structure that provides a high throughput indexing for the key-value store. Traditional LSMs are hierarchical with the data stored in memory and disk. Further, both the memory and disk data structures contain multiple levels of hierarchy. The multi-level hierarchy makes these approaches a natural fit for heterogeneous memory, where each level can be placed in one or more memory types. For exploring LSMs, we use LevelDB, an open source on-disk key-value store based on LSM, whose design is related to Google's Bigtable [50] database system. The data structures of most current LSMs implementations are designed for systems with a single-level memory and disks. However, using the disk-based design cannot extract maximum benefits for future heterogeneous memory systems with deep memory hierarchy due to the much higher throughput and lower latency expected of heterogeneous memory technologies such as NVMs. As we show in our analysis, when using fast NVM-based storage, which is expected to be 100x faster that block-based SSD, current LSMs achieve less than 2x to 3x improvement in write and read throughputs. We explain these significantly lower than expected benefits to the following factors.

- Lack of in-memory storage data structures - Current designs lack the capability to exploit byte addressability of future storage technologies. The data structures aim to reduce random access by preparing data for sequential access, and as a result, the time spent on the preparation (e.g., serialization) dominates the cost compared to the benefits from sequential access.

- Lack of parallelism - Current designs of LSM-based key-value stores lack the capability to extract parallelism from low latency high bandwidth storage technologies.

In the remainder of this section, we briefly discuss LSM trees and LevelDB – a popular LSM tree implementation from Google, and then discuss our HeteroDB design.

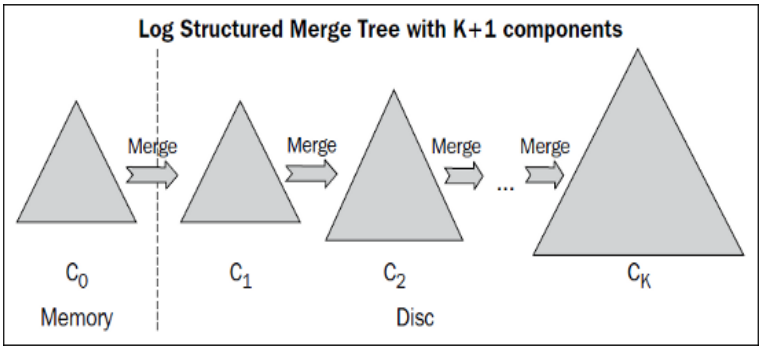


Figure 77: Structure of LSM trees.

7.3.1 Background

7.3.1.1 Log-structured merge trees

An LSM-tree [115] is a persistent structure that provides efficient indexing for a key-value store. LSM trees and LevelDB aim to achieve higher write performance by staging data across multiple levels, thereby avoiding random writes to block-based devices, unlike other persistent-friendly data structures such as B-trees [77, 32]. Figure 77 shows a high-level view of an LSM tree with multiple trees (or components) C0 to Ck, with C0 in the memory, and C1-Ck on the disk. In LSMs, generally, the levels have an increasing size. For example, in LevelDB, each level is at least ten times larger than the previous level. Keys are first inserted into memory level C0, and are slowly trickled down to disk-based data structures. Most LSM trees use B-trees for the disk-based data structures. However, buffering persistent data in the memory does not provide durability in the event of application or power failure. Hence, to avoid data loss, LSM implementations append the data to a read-ahead log before inserting to the memory-level. For search operations, LSMs require iterating across each of the levels (components), by proceeding first with level C0 containing the most recently inserted values, followed by other levels. As the size of levels increase, the search time also increases. Hence, point queries have higher latency and overall throughput with LSM. In general, LSMs are update-friendly, and read operations are slower compared to other alternatives for NoSQL designs [140, 155].

7.3.1.2 LevelDB

LevelDB [74] is an LSM-based NoSQL key-value store derived from Google's Bigtable implementation, and is widely used from browsers to datacenter applications. Apart from simple key-value operations, LevelDB provides range queries and also provides compression of data using the Snappy compression we discussed in earlier chapters. The LevelDB design maintains two levels of in-memory data structures known as "memtable" for buffering data, and seven levels of disk-based sorted string tables (SSTable). The first memtable contains a mutable skip list whereas the second contains an immutable skip list. A compaction thread (background thread) is responsible for converting the immutable skip list to SSTable compatible data during compaction. Compaction is initiated when a capacity threshold of mutable memory is reached. All the levels except the memtable are mutually exclusive and do not maintain redundant data. Although LevelDB allows multiple threads to simultaneous fetch data, every 'Get' (also referred as read elsewhere) operation is single threaded, irrespective of the number of levels on the disk.

7.3.1.3 SSTable

An SSTable [104, 50, 74] provides a persistent, ordered immutable map from keys to values, where both keys and values are arbitrary byte strings. Operations are provided to look up the value associated with a specified key, and to iterate over all key/value pairs in a specified key range. Internally, each SSTable contains a sequence of blocks (typically each block is 64KB in size, but this is configurable). A block index (stored at the end of the SSTable) is used to locate blocks; the index is loaded into memory when the SSTable is opened. A lookup can be performed with a single disk seek: we first find the appropriate block by performing a binary search on the in-memory index, and then read the appropriate block from disk. Optionally, an SSTable can be completely mapped into memory, which allows us to perform lookups and scans without touching disk. Prior work such as as Wisckey [104] have explored methods for optimizing LSMs for SSD. Wisckey specifically focusses on improving the performance by reducing the number of reads or writes required to perform a single key-value operation, also known as the amplification factor.

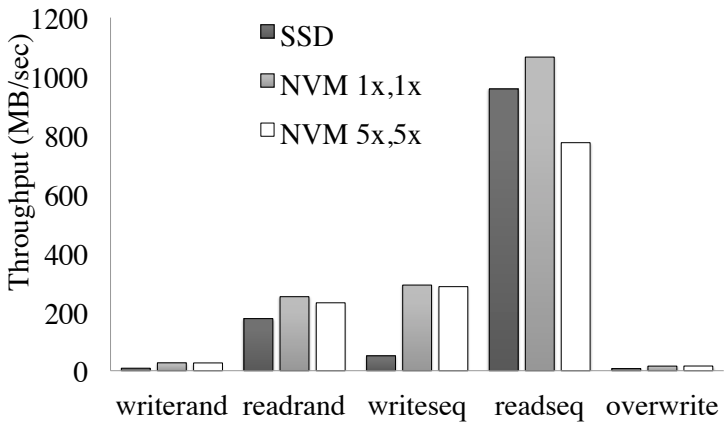


Figure 78: SSD vs. NVM throughput for LevelDB benchmark, 1 million keys, 4KB value size.

7.3.2 Motivation

To understand the performance implications and to identify opportunities for extending LSMs for heterogeneous memory, we evaluate the LevelDB's LSM implementation. We use the widely-used LevelDB benchmark. For our analysis, we use the heterogeneous memory emulation discussed in prior chapters. We first analyze the benefits of using SlowMem (NVM) compared to SSDs, followed by investigating the sensitivity to other critical parameters such as the size of the mutable memory table. For all our analysis, we run the benchmark for 1 million keys.

7.3.2.1 SSD vs. SlowMem

Figure 78 compares the throughput (in MB/sec) of different access patterns for 4KB value size when running with SlowMem and SSD. For SSD we use a SATA-based Intel-520 series commodity SSD. When using the NVM, all SSTables are placed in SlowMem, whereas the mutable and immutable memtables are placed in FastMem. The SlowMem 1x, 1x bar represents the performance when the bandwidth and latency of SlowMem are same as FastMem – an ideal best case performance when using the vanilla LevelDB. SlowMem 5x, 5x represents the case with 5x latency increase and 5x bandwidth reduction.

Observations. First, as shown in the figure, the SlowMem as excepted delivers a significant gain for sequential writes compared to SSD. Surprisingly, the benefits from random

updates are low. This result is against the general wisdom that for low-latency, high bandwidth byte addressable memory technologies such as NVM, random and sequential access are expected to be same. However, a close examination reveals that for random updates, the cost of random inserts into the mutable memory table skip list is almost 3x higher compared to the cost of sequential update skip list. For the sequential insert operation to skip list, the next pointer is often available in the processor cache compared to the random skip list insert, where data has to be fetched from memory by iterating across other entries from the top row of the skip list to the lower row. Next, regarding the sequential and random read performance, the performance of SSD vs. NVM-based access is almost same. This is because, for Level-DB, the reads happen from a memory-mapped DB. Consequently, most of the read operations occur from the buffer cache. When increasing the number of unique keys in the database or using larger value sizes, the number of buffer cache hits can reduce. In such cases, NVM with its higher bandwidth is expected to provide higher throughput gains compared to SSD.

7.3.2.2 Impact of memtable size

SSTables are disk friendly structures. Before moving data from the memtable to SSTables for compaction, the data is serialized from the memtable to SSTable format, whereas for reads, the data is deserialized. Using a large memtable reduces the frequency of compactions, thereby reducing the impact of background compaction on applications. Hence, to understand the impact, we increase the memtable size and analyze the throughput and latency. In Figures 79 and 80, the x-axis shows the memtable size, and the y-axis show the throughput in MB/sec, and the latency in microseconds per operation.

We make the following observations. First, the sequential and random write performance improves mainly because with increasing memtable size, the stalls suffered by writes and overwrites due to compaction are lower, primarily because of the reduction in the compaction frequency. Next, regarding the random and sequential read performance, interestingly, the throughput reduces and the latency increases with the increasing memtable size. This is because skip lists have an average search complexity of O(log n). As the

memtable size increases and the depth of the length and height of the skip list increase, the time to iterate and find the required key increases. The overhead of skip list traversal for random access is at least 3x higher compared to the sequential access. For a sequential linked list, most elements in the iteration path (the previous elements) are already present in the process cache from the prior iterations, unlike random access where a dominant number of elements in the iteration path requires memory access. In contrast, the data stored to SSTables is sorted during compaction. Also, a bloom filter with O(1) complexity is used to identify the level in which the requested key is possibly present. Hence, access from SSTables placed in NVM can be faster than a large skip list. These results show that although increasing the memtable size can improve the write and update performance, for read performance, to avoid traversal costs, just increasing the memtable size buffer is insufficient.

Summary. Current LSMs have been designed to reduce the impact of slow block-based disks by tiering data across memory and storage. Importantly, tiering improves write performance because updates/writes are buffered in memory before they are written to disk. Using heterogeneous memory such as NVM for current storage-efficient LSMs can improve the performance, but the gains are significantly lower compared to the 5x-10x higher bandwidth and an order of magnitude lower write latency of NVM. Further, using NVMs improves write performance, but the read access throughput and latency gains are less than 50%. Hence, with HeteroDB, we aim to redesign LSMs for heterogeneous memory and to improve both write and read access performance.

7.3.3 HeteroDB design goals

HeteroDB's design aims to exploit the high bandwidth, low latency, and large capacity of future heterogeneous memory to improve the performance of LSM-based NoSQL databases. Unlike traditional LSMs that improve write/update performance but compromise read/-query performance, our design aims to address the read and the write performance. Based on this overall goal, we formulate the key design aspect of HeteroDB.

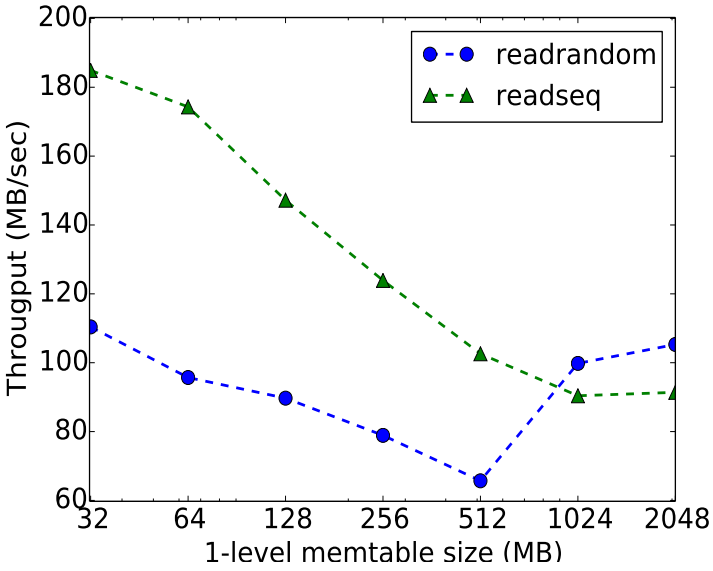


Figure 79: Memtable size vs. throughput.

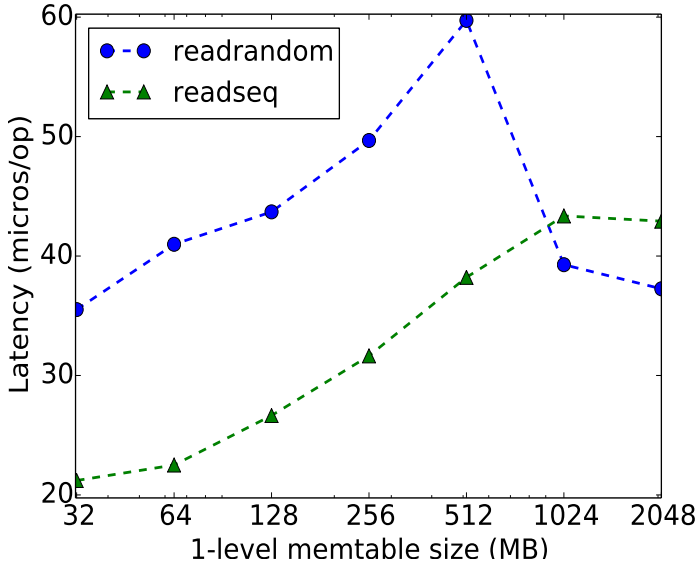


Figure 80: Memtable size vs. latency.

- Redesign persistent data structures: Current LSM designs contain one level of in-memory skip list-based memtable followed by disk-based SSTables designed to reduce random access to block-based devices. However, in heterogeneous memory such as NVM, the difference in the cost of random and sequential access is significantly lower or negligible compared to the block-based device. Hence, our design aims to redesign data structures that can take advantage of the memory-based byte addressable access, while also providing the persistence guarantees of existing LSM-based structures such as SSTables.

- Exploiting parallelism: Unlike block-based storage devices, heterogeneous memory can provide up to 10x higher bandwidth and an order lower latency. Hence, to saturate the bandwidth at low latency, it is important to utilize the CPU parallelism across several levels of memory and storage hierarchies. We propose a design that supports concurrency at finer granularity, parallelizes read operations, and consequently improves the overall throughput.

7.3.4 Design and implementation

Based on these design goals, we next discuss the details of the design and implementation. Our implementation extends the LevelDB design with support for heterogeneous memory, but we believe that our extensions are generic and applicable to other LSM implementations such as Facebook's RocksDB [3], MongoDB [15] and others. Figure 81 shows a high-level design of HeteroDB.

7.3.4.1 Exploiting byte addressability with NVM-immutable memtable

As discussed extensively throughout this dissertation, future byte-addressable memory-based storage technologies such as NVMs have properties more similar to memory than to disk. However, current LSM implementations use persistent data structures such as SSTables that have been specifically designed for block-based storage devices to reduce random access across critical paths of the key-value stores. The data is moved from an in-memory memtable to SSTable during a background compaction. But moving the data

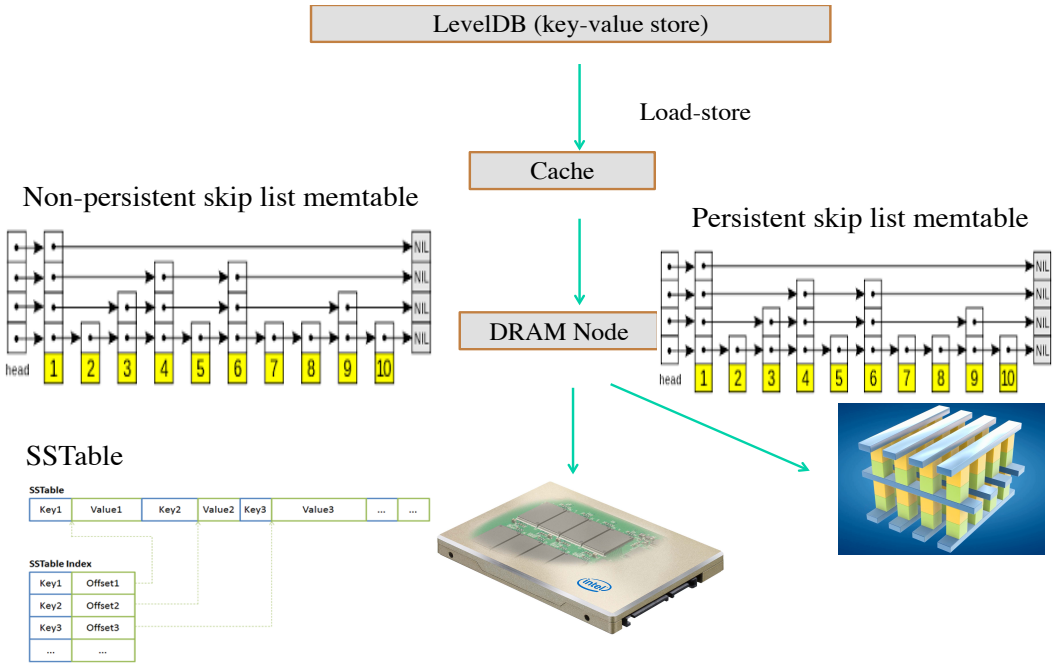


Figure 81: High-level design of HeteroDB.

requires first serializing the data from an in-memory skip list format to a disk-based block format. Next, before inserting a key-value pair to any level of the SSTable, the keys must be sorted to reduce random disk access during a read or an update operation. However, both serialization and merge sorting are expensive, and can propagate across multiple levels of SSTables. Consequently, the background compactions stall the foreground operations until free space is made available at the mutable memtables. The write and update performance suffers from frequent stalls due to forced compaction, whereas the read performance primarily suffers from the cost of serialization and scan across multiple levels of SSTables during the read operation. Therefore, using just one-level of DRAM-based memtable and having SSTable structures in NVM does not deliver adequate advantages of using NVM, and in fact, can significantly impact on both write and read throughput and latency.

HeteroDB addresses this by introducing an NVM immutable (referred as NVM-immutable) memtable between the DRAM-based memtables and the SSTables in the LSM hierarchy. The NVM-immutable memtable is larger in capacity compared to the mutable memtable, and also provides the same persistence guarantees provided by the SSTables. Currently, the NVM-immutable only replaces only one level of the multi-level SSTable. Figure 81 shows

a high level design. The NVM-immutable table uses the same skip list structure but also provides persistence. In LevelDB, both the mutable and immutable memtable reside in the DRAM and have the same size. When the mutable memtable is full, it is made immutable, and a new mutable memtable is allocated for adding keys. A background thread has to compact (move) contents of the immutable table to the SSTable before the mutable memtable is full. Failure to complete compaction stalls new insert to the mutable memtable. In HeteroDB, the compaction happens from the DRAM-based immutable memtable to the NVM-immutable table. Using an in-memory skip list data structure as opposed to the SSTable structure provides the following benefits.

- Compaction of data between DRAM and NVM-immutable memtable does not require serialization of the skip list nodes to SSTable blocks. Consequently, the background compaction is faster and reduces the wait time for new updates to the mutable memtable.

- Compaction involves inserting from the skip list to the other 'larger' skip list in the NVM-immutable memtable. Using an in-memory data structure exploits the low-latency byte addressable random access of NVM and avoids the need to read and write an entire block of data as required in SSTables.

- Finally, retaining a unified skip list-based memtable structure in the NVM enables HeteroDB to keep most functionalities and optimization of the existing LevelDB memtable design.

7.3.4.2 Persistent skip list for durability

Just replacing an SSTable with an NVM-immutable memtable is insufficient. SSTables provides durability and also limit the size of the log. During any insert or data update operations, the insert operation is first added to a persistent log before inserting the key-value pair to the mutable memtable. The log is truncated and cleaned after compaction of the in-memory memtables to the persistent SSTable. HeteroDB, by extending the memtable structure to NVM must also provide the same level of durability guarantees as provided

by the original LevelDB design. Hence, HeteroDB proposes a persistent skip list for the NVM-based immutable memtable.

Key idea. In a persistent skip list, all nodes are mapped to a large contiguous memory mapped region in NVM. Each skip list node points to the next node using a physical offset relative to the starting address of the mapped region, as opposed to using a virtual address. The offset of the root node of the skip list from the starting address of the mapped region is logged separately. In the event of a failure, the persistent region is remapped, and using the offset of the root node of the skip list all other elements are recovered.

Implementation. To implement the persistent skip list, we first extend the LevelDB memtable with a custom NVM allocator that allocates a large region of contiguous pages, and also maintains the metadata of the allocated region. For the allocator, we use our persistent object allocator previously discussed in Chapter 3. Each node in the immutable NVM maintains a physical offset pointer which are updated during the insert operation. Note that adding nodes for both existing SSTable and NVM persistent memtable does not require a transactional update because as discussed earlier, a transactional log is already updated during memtable inserts. Traversal of a persistent skip list during recovery involves iteration of offset, and using a persistent skip list does not impact performance.

7.3.4.3 Parallelism in read operations

The number of cores per machine and in a socket is consistently increasing across the datacenter and HPC servers, as well as in end-user devices. However, low device bandwidth and high latency in block-based devices such as SSD limit the parallelism in the storage-intensive application. In contrast, heterogeneous memory provides the capability to extract parallelism at the hardware-level by reducing the device access latency from few milliseconds to tens of nanoseconds. For utilizing these hardware capabilities, a software-level redesign is also important. Although current LSM implementations such as LevelDB enable multiple threads to perform simultaneous data lookup, each lookup operation across multiple levels – memory and storage-levels – is sequential. As a result, the latency increases with increasing the levels of memory and storage hierarchy.

HeteroDB addresses this drawback by parallelising lookup operations. During every lookup, different threads simultaneously query their affiliated memtable or SSTable structure, thereby, utilizing the significant DRAM and NVM bandwidth. Parallelism presents two challenges. (1) The lookup operation can vary from few microseconds to hundred's of microseconds depending on the data size and the size of the database. Hence parallelism introduces additional thread management overhead. The cost of creating new threads and synchronization overheads can be more expensive for read operations that require less than 5-10 microseconds even with non-parallelized search. (2) Although the immutable memtables are mutually exclusive, a single key with the latest version in mutable and an older version in immutable tables can exist simultaneously. Using a wrong version impacts the correctness.

In our current design, to address the first challenge, we implement a worker thread pool with a dedicated thread affiliated with one or more levels of memory and storage. One dedicated thread is created for the mutable memtable, one each for the immutable DRAM and NVM memtables, and the main parent thread is used to search across the SSTables. Note that all the child thread creation happens during the database initialization and the child threads except the parent thread are in an inactive state until a new read request is added to the work pool. Access to the work pool is synchronized, and the parent waits for the child threads to complete. Next, to address the second challenge for maintaining correctness, the mutable memtable version takes precedence over other levels. Once a key is found, HeteroDB suspends NVM and DRAM immutable memtable and SSTable threads except the mutable memtable thread. Our results show the benefits of exploiting parallelism across every read access in terms of higher throughput.

7.3.5 Evaluation

We present evaluation results that demonstrate the benefits of the HeteroDB design. Our evaluation goals are to understand the following:

- Persistent memtable – We first evaluate the benefits and implications of introducing a persistent NVM-immutable memtable apart from the existing persistent SSTable.

- Exploiting parallelism – NVMs provide low latency and high bandwidth, unlike disk. Hence, we explore how parallelism across multi-level memtable and SSTable structures can benefit the read performance in LSMs.

7.3.5.1 Benchmarks and methodology

For the evaluation, we use the popular 'db_bench' benchmark developed with LevelDB. To limit the total database size to fit into SlowMem emulated with one NUMA socket memory, we limit the total number of keys to 1 million, with a key size of 16B, but we vary the value sizes in our experiments. LevelDB by default uses the Snappy compression of database. We turn off compression for easier understanding of the results.

For our experiments, we use an Intel(R) Xeon(R) CPU X5560 @ 2.80GHz processor with 32GB of memory running the Linux 3.9 OS. We use the PMFS-based filesystem with NVM (SlowMem) emulation for storing the SSTables.

7.3.5.2 Impact of persistent NVM-immutable memtable

Figure 82 shows the benefits and implications of the HeteroDB's persistent NVM-immutable design. The x-axis shows the access patterns, and the y-axis shows their corresponding throughput (MB/sec). Adding an NVM-immutable table improves the random access (read, write, and update) performance significantly, but the sequential access performance reduces compared to the vanilla 1-level LevelDB. We next discuss the reasons for this behaviour. We use NVM-immutable size of 1GB, which is 10x larger than the mutable DRAM memtable. This complies with the LevelDB's design of using 10x larger storage when descending down the storage hierarchy.

Random writes and reads. In the case of vanilla LevelDB, for random writes, the key-value is first inserted to the skip list-based memtable. When the memtable is full, the data is compacted to the lower SSTables. Note that, during compaction to SSTables, the keys must be sorted before storing. Random writes incur higher sorting cost with an $O(n * logn)$ complexity. Sorting also happens when the first level SSTable is full, and data is merged iteratively from higher to lower levels. Specifically, for random keys, the sorting (merging) cost increases the background compaction time. Consequently, this stalls

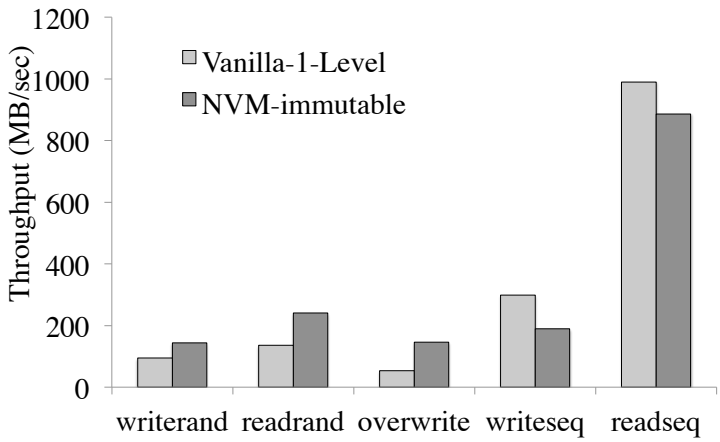


Figure 82: Impact of NVM immutable vs. vanilla 1-level memtable.

new inserts to the memtable until the DRAM immutable level is completely freed. In contrast, with HeteroDB's NVM-immutable design, during compaction, the keys in the DRAM immutable memtable are iteratively inserted into the NVM-immutable skip list-based memtable with 10x larger capacity. The NVM-immutable table improves the write performance due to two main factors, (1) the compaction to the NVM-immutable table involves inserts into the NVM-immutable skip list is $O(logn)$ compared to the SSTable's $O(n * logn)$ complexity. Next, (2) the inserts into the NVM-immutable skip lists does not require serialization unlike the vanilla LevelDB, where each skip list node is serialized into the SSTable format. Eventually, when the NVM-immutable memtable runs out of free space, background compaction to SSTable is required, but the memtable to SSTable compaction frequency is reduced with the large NVM-immutable memtable design. Regarding the random reads, with a 1GB NVM-immutable, higher number access hits from the immutable memtable avoids the need to lookup the SSTable frequently. However, the gains are marginal mainly because, as discussed earlier, although the read request hits in the NVM-immutable avoids deserialization from the SSTable format, random access of skip list incurs additional overhead. We will shortly discuss the benefits of parallelizing access across each LSM level to hide the skip list traversal cost. Finally, for the updates (overwrites) to the existing key, first, the key and its corresponding value have to be read followed by a write. The benefits of NVM-immutable's random write and read performance combine of random write and

read improvements combine to provide 3x gains.

Sequential writes and reads. For sequential access of both writes and reads, the cost of insertion to the skip list and the cost of compaction in the SSTables reduces considerably. This is mainly because, for the skip lists, during sequential inserts or reads, most elements along the list traversal path are processor cache hits avoiding the access from memory. Next, during sequential inserts, the sorting and merging cost significantly reduces the compaction cost across multiple SSTable levels. Further, for read operations, data from SSTable is read in the granularity of blocks. Hence, for the sequential reads, except the first key-value pair, all the key-value pairs in a block are read from the buffer cache. In contrast, the NVM-immutable design, by adding one more level of skip list traversal increases both read and write cost by $O(logn)$, thereby reducing the sequential access performance compared to the vanilla design. *The results show that introducing the NVM-immutable design improves the random access performance by 3x, whereas the sequential write and read throughput deteriorates by 57% and 11% respectively.* Although the random access gains are significant, the sequential access performance reduction must be addressed.

7.3.5.3 Parallelism in read access

We next discuss the impact of increasing the read access parallelism in HeteroDB by increasing the number of threads per operation. The number of threads per read operation is varied by parallelizing reads across mutable, immutable, and SSTables. To understand the impact we evaluate the sensitivity of the number of threads per read operation and also vary the value sizes. In Figures 83 and 84, the lines labeled serial show the serial read operations without threading, '2-threads per read' represents one thread to read all the memtables (including mutable and immutable tables), and one thread for searching the SSTables. '3-threads per read' represents one thread each for the mutable, immutable, and SSTables respectively. Finally, for the '4-threads per read' case, one thread is used to search across the mutable memtable, one thread each is used for the DRAM and the large NVM immutable tables, and one thread (main thread) is used for the SSTables.

In Figure 83, the values on the x-axis shows the value sizes, ranging from 1KB to

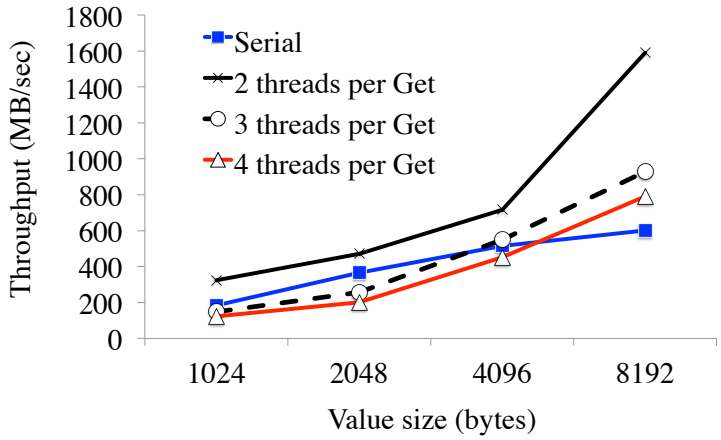


Figure 83: Random access - Threads per Get vs. value size.

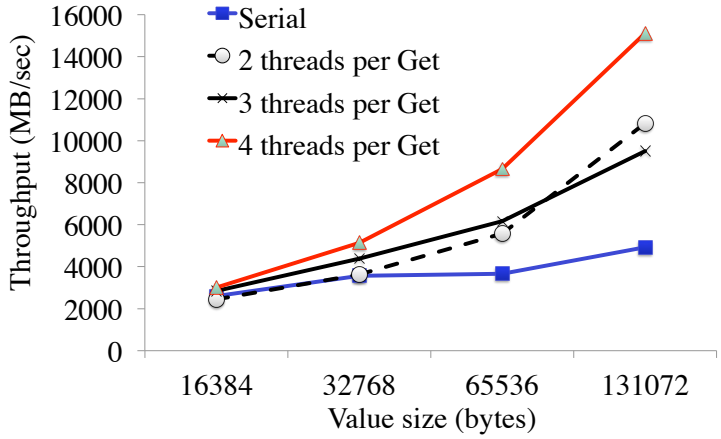


Figure 84: Threading impact for large value size.

8KB, and the y-axis shows the corresponding throughput. At a high level, the benefits of parallelism are better realized when the latency per operation increases either due to larger value sizes or due to the database size. As observed in the figure, using multiple threads (2-threads, 3-threads, and 4-threads) per read operation show higher throughput for value size larger than 4KB compared to the serial access. However, for value size smaller that 4KB, the 2-threads case performs better than all the approaches, but the 3-threads and 4-threads show lower benefits even compared to the serial case. The reason for this behavior is as follows.

As discussed in the HeteroDB design, conflict-free worker thread pool management and assigning tasks to threads require expensive synchronization and communication. For

smaller values up to 4KB, even without threading, the latency per operation is less than 15 microseconds as most of the reads are either from the memtables or higher-level SSTables. However, the cost of thread pool management for the 3 and 4-threads cases dominates the small benefits from parallelism, resulting in a lower performance compared to the serial access. In the 2-threads case, one thread looks up all memtables and the main threads looks up the SSTable and no thread pool synchronization or communication between the threads are required. Hence, the 2-thread case can exploit the maximum benefits from parallelism. As the value size increases, the latency per operation increases, as most accesses require search across lower levels of the LSM. For values larger than 4KB, the 3-threads and 4-threads case provide higher benefit compared to the serial version, but still, 2-threads without any thread pool management cost achieves the maximum benefit. This also explains the small benefit of 3-threads case compared to the 4-threads case.

To validate this observation, Figure 84 shows the throughput for large values ranging from 16KB to 128KB. For large values, we use a total of 200K keys compared to 1 million keys in the previous evaluation for small values. We use a lower number of keys because our NVM emulation platform evaluation platform has a 16GB of per-NUMA node memory, and we store in it the NVM-immutable, the SSTables, and the log. Note that the throughput scales for large values are different compared to the scales for small values. As the size of values increase, the throughput also increases due to better use of bandwidth. For larger values, the latency per operations is in few tens of microseconds. Hence, the 4-threads case outperforms the 2-threads and 3-threads case. The overall read access bandwidth increases by 3.5x compared to the serial access. The poor performance of 2-threads case for 8KB and 16KB value size is because of using smaller number of keys due to the platform restriction, and most of the accesses occur from the higher levels. For our future work, we plan to evaluate in a larger NVM capacity platform. *These results show that for the optimal use of high bandwidth, low-latency heterogeneous memory such as NVM, it is important to exploit parallelism unlike disks, where the device bandwidth significantly limits parallelism.*

7.3.6 HeteroDB summary

To summarize, we discussed the performance benefits of redesigning LSM-based key-value store such as LevelDB to exploit the high bandwidth, low latency, and large capacity of future heterogeneous memory. We discussed and developed byte-addressable persistent data structures and methods to increase the access parallelism. We also evaluated our proposed techniques which improved write and read throughput by up to 3x. We believe more opportunities exist in improving the LSM performance for heterogeneous memory including fundamental improvements in memory and disk data structures.

CHAPTER VIII

RELATED WORK

Prior research on heterogeneous memory can be categorized into hardware, system software, and application-level support. We first discuss the hardware-level support for heterogeneous memory, followed by OS-level capacity scaling and data placement. We then discuss using heterogeneous memory such as NVM for persistence and the implications of persistence.

8.1 Heterogeneous memory for capacity scaling

Hardware support. A significant body of hardware research have explored two promising heterogeneous memory technologies – byte addressable NVMs such as phase change memory (PCM), and on-chip 3D-DRAM. Apart from using NVM for persistent storage, several NVM research [101, 125, 98, 128] have discussed the benefits of using them as an extended memory when attached a memory controller. Regarding stacked 3D-DRAM, several proposals have explored their use as a large last level (L4) process cache to reuse and extend existing [163, 92, 123, 102, 53] existing cache replacement policies. In contrast, other research [52, 64, 31, 80, 107, 119] have highlighted significant hardware change (cache controller redesign, tag space, lack of application flexibility) and favor using them as a high bandwidth DRAM managed by the software.

Capacity scaling. Prior research [125, 60, 137, 98] has used NVM as an alternative to DRAM with transparent page replacement strategies between DRAM and NVM. Qureshi et al. proposed a hardware-based model that treats DRAM as a cache and NVM as main memory, using the page access patterns in the OS/hardware to move data between DRAM and NVM. While this provides application and OS transparency, copying data between a fast and slow memory can add significant overhead. It limits application flexibility in memory placement, and more importantly prevents the use of persistence property. Saxena et al. [137] propose using Flash as an extended memory managed by the VFS, and use several data prefetching techniques. PMFS also enables extending memory capacity via a

VFS. In contrast, our pVM design discussed in Chapter 3 provides a VM-based management and, by treating NVM as a NUMA node, allows for seamless memory scaling and flexible memory placement mechanisms. To the best of our knowledge pVM is the first OS design that considers extending the VM subsystem for both capacity and persistent storage, and that provides flexible memory placement policies to applications.

Software and hardware-level data placement Batman [52] proposed a H/W-based data placement methods focused on increasing the cumulative DRAM and Stacked-DRAM bandwidth using a randomized FastMem- SlowMem for applications. X.Dong et al. [64] propose memory controller redesign to optimize data migration using a specialized H/W address translation that maps FastMem slots with with page physical address. Meswani et al. [107] modify TLB to find hot pages. In contrast, HeteroOS is a purely OS-based solution, and prior hardware solutions can complement its performance.

OS management of heterogeneity: Prior heterogeneous memory research such as [35, 68, 119] use different forms for offline memory classification. Phadke et al. [119] categorize application data structures into latency, bandwidth or CPU-intensive and use the analysis for OS-level page allocation, whereas, Dulloor et al. [68] use the analysis for library allocator-level placement. In contrast, our HeteroOS design discussed in Chapter 4 is application transparent and does not require any static analysis.

Next, Meshwani et al. [107] consider only heap-intensive applications. In contrast HeteroOS discusses the needs an manages heterogeneous memory for memory, CPU and storage-intensive datacenter applications. We have extensively discussed the drawbacks of VMM-exclusive HeteroVisor. To best of our knowledge, HeteroOS is the first design to manage memory heterogeneity at OS-level for both non-virtualized and virtualized environments with coordinated (guest-VMM) management. In fact, except few commercial products [26] and academic projects [131], most VMMs do not expose even the NUMA topology to guest VMs. Unlike NUMA, significant memory property differences require a separate heterogeneity-aware management as discussed by [107] which the HeteroOS design provides.

8.2 Persistent storage in NVMs

We next discuss the persistence related heterogeneous memory research.

Filesystem for NVM. Prior research such as Aerie [145], Moneta-D [48], and some early work such as FUSE [127] aim to reduce the OS overhead by moving the filesystem to the user space, and directly using the hardware for read/write operations. Other research [47, 154, 67, 57] has proposed optimizing current block-based filesystems to adapt them for memory-based persistence, as explained earlier. Our system pVM, borrows several OS consistency and durability-related ideas from such prior work, and adapts them for object storage. A possible future work is to consider the co-existence of a filesystem with pVM's object based storage.

NVM as a heap. The benefits of using NVM as a heap for persistent storage have been well documented by the earliest systems – Rio Vista [103], and RVM [136]. These solutions provide a heap-based persistence and OS-level management that uses a combination of block-based storage and battery-backed RAM. In addition, Rio Vista provides persistence guarantees only at log flushes. Recent research such as [146, 56] also offers persistent data programming models, in addition to the capabilities of RVM. pVM's user-level library borrows ideas from these proposals. However, the key contribution of pVM is the novel extension of the VM subsystem for achieving both memory scaling and persistence. [146, 56] depend on VFS-backed memory, and we have already discussed their limitations. Furthermore, Volvos et al. [146], and SCMFS [154] propose an NVM persistence OS zone. pVM extends such ideas to create capacity and persistence zones, but more importantly, it provides a cleaner NUMA-based abstraction for NVMs, and provides users with flexible memory placement policies. Guerra [78] et al. discuss solutions to prevent mixing persistent and nonpersistent pointers via memory protection, which can also be adapted to pVM.

Durability and consistency. NVM hardware-software must support required consistency and durability across application sessions. When using the processor cache to hide write latency, since cache data can be evicted in any order, to maintain ACID properties, prior efforts have used write-through caches [146], or epoch-based cache eviction methods [57] using memory barriers to order NVM writes. Further, durability can be affected during

power failures or device crashes, leaving the application in a non-deterministic state, e.g., due to partial updates (note that both data and metadata must be saved consistently). A common approach to deal with this relies on application commits, which in turn trigger cache flushes. Although sufficiently frequent flushes can reduce the possibility of non-recoverable failures, additional transactional mechanisms are needed for atomicity, accompanied with logging (e.g., undo/redo) support for durability. A recent work [160] proposed hardware-based nonvolatile cache and nonvolatile memory to enable multi-version support with in-place updates (avoids the logging cost). The cache contains the dirty version and the memory contains the cleaner version. While such micro-architectural changes can reduce the cost of logging, we focus on the software optimizations for existing hardware (volatile cache). In this thesis, we discuss methods to provide atomicity and durability guarantees with efficient use of cache.

Relaxing persistence. To reduce persistence overheads, Pelly et al. [117] propose a consistency model that allows reordering between one or more independent transactions for better concurrency. Reordering is done only for writes to the cache, whereas writes to the NVM from the cache are ordered. This approach constitutes a variation of the epoch-based consistency model that requires hardware and software changes for reordering writes. Y. Lu et al. [105] propose a loose ordering consistency (LOC) protocol for relaxing intra- and inter-transaction ordering. In their approach, the log area is divided into blocks of 64 bytes with one metadata header for seven blocks. Committing log data in block-groups avoids repeated metadata update. After a failure, the consistency is validated using the group-level metadata. LOC assumes the presence of a nonvolatile cache. Next, [162] improves persistent storage performance by adding intelligence to the memory controller for prioritizing persistent data updates with non-persistent data updates. All the above proposals require hardware and software modification. A recent software-only research, NVRAMDB [118], uses NVM as a disk for page-based persistence, and proposes performance optimizations specifically for databases. NVRAMDB uses a batched/group commit that first copies the original data to an UNDO rollback/recovery log, then buffers multiple transactions in a DRAM buffer, and finally commits all buffered data to NVM. Although this approach

improves the throughput, it does not change the total data logged to NVM. In fact, by buffering transactions in DRAM and then copying to NVM, this solution consumes additional CPU and DRAM energy. In contrast, we propose a group commit update customized for Mem-persist that reduces energy use.

Reducing energy overheads of persistence. To our best of the knowledge, the energy aware persistence (EAP) discussed in this thesis is the first to reduce energy overheads of persistence. EAP differs from the work reviewed above [117, 105, 162, 118] in several ways. (1) EAP identifies and addresses ACID software components that increase energy. (2) Other than NVM support, EAP does not require additional hardware changes. (3) Prior work principally seeks improvements in application performance, a case in point being the lazy asynchronous, or relaxed atomicity and ordering in [105] and [117]. Using such methods change when certain actions are taken, but they do not reduce the total CPU instructions or NVM accesses. As a result, these proposed methods do not directly reduce the persistence-related energy consumption. (4) EAP's relaxed durability model is designed for Mem-persist where it is important to classify data and metadata for maintaining correctness of heap and application state, unlike [117]. Finally, (5) our mechanisms are driven by the current energy availability, with flexibility to switch between performance and energy-efficient modes.

8.3 Applications of heterogeneous memory

We next discuss several prior research that use heterogeneous memory such as NVM for improving application performance and reliability. We specifically focus on applications such as checkpoint, browsers studied in this thesis.

8.3.1 Checkpoint-restart

In Chapter 7, we discussed the benefits of using NVM as a virtual memory for HPC checkpoint-restart capability.Several prior research have explored checkpoint-restart using NVM. We next discuss them in detail.

Multilevel checkpointing. Multilevel checkpoints (node local and remote) have been

extensively studied in various forms like diskless [120, 165, 110, 37] or local storage checkpointing. To our knowledge, Plank et al. [120] were the first to propose a diskless checkpointing in distributed applications by using additional processor and memory to replicate checkpoint data without relying on stable storage. To reduce memory usage, several alternatives, like erasure coding technique [121], compute node pairs for in-memory checkpointing [165], or use of a RAMDisk-SSD hybrid approach for reducing memory consumption have been proposed. More recently, Moody et al. [110] made an extensive failure modeling of multilevel checkpoint using a Markov model and further built a multilevel checkpoint library using RAMDisk or SSD for a local checkpoint. As discussed in Chapter 7, our work complements prior multilevel checkpointing research, but our novel contribution is understanding the benefits and implications of heterogeneous memory such as NVMs for multilevel checkpoints. Further, we show that just replacing HDDs or SSDs with NVM will not be sufficient to exploit its hardware capabilities. By using NVM as a virtual memory, checkpoint performance can be substantially improved.

NVM checkpointing. Most prior work on NVM checkpointing has focused on transparent checkpoint and architectural enhancements. Dong et al. [63, 62] propose a local and global checkpoint approach for NVM based transparent checkpoint, and for addressing limited NVM bandwidth, they propose a stacked 3D DRAM-NVM design. Although such future hardware is complimentary, in Chapter 7 we show that such issues can be addressed from software approaches too. Finally, Li et al. [99] show that directly exposing NVMs for simulation and checkpointing slowdown application by up to 25%. Our initial analysis using binary instrumentation showed similar results, and hence we use a shadow buffering mechanism.

8.3.2 Sandboxing and Browsers

The impact of software-based sandboxing has been extensively studied, from the seminal work of Wahbe et al. [148] and most recently by [156]. Recent work on browsers focused on the complete browser and OS redesign for security [144], but lacks support for current browser frameworks. To the best of our knowledge, our work is first of its kind for exploring

the opportunities of using heterogeneous memory such as NVM for reducing the sandboxing impact on storage. Approaches such as Android [42] focus in moving a major portion of sandboxing to the OS, similar to, but for OS-agnostic applications like browsers, completely relying on OS-based isolation seems unlikely. Finally, recent work on exception-less system calls [142] studies the impact of reducing system call blocking costs and we believe such kernel techniques can be very useful in our future work.

8.3.3 Log-structured merge trees (LSM)

Prior work such as VT-trees [140] and LSM-trie [155] have focussed in extending and optimizing the traditional LSM data structures originally proposed by O'Neil et al. [115] to improve read and write throughput, and also reduce redundant data writes and reads. VT-tree avoids sorting previously sorted key-value pairs during compaction thereby reduces unnecessary read and writes across SSTables. LSM-trie uses a trie data structure to organize keys and proposes a more efficient compaction based on the trie. Other research such as Wisckey [104] and LOCS [151] focus on improving the LSM's for modern flash storage. Several prior works such as IndexFS [132] has extended LSMs for application-specific usage such as the object-based filesystem. NVMKV [106] proposes flash-aware key value store whereas ATLAS [97] designs a distributed key-value store using LSM. To the best of our knowledge, HeteroDB is the first work to explore the redesign of LSM data structures for the heterogeneous memory-based storage device such as NVM.

CHAPTER IX

CONCLUSION AND FUTURE WORK

This dissertation covers multi-dimensional aspects of memory heterogeneity. We first summarize the dissertation's key contributions and then discuss the future work.

9.1 Conclusion

Digitization of universal data is increasing at an astonishing pace. Experts predict a 4300% increase in the annual data generation. However, fundamental computing elements required for processing the data are lagging behind. Recent predictions show that processor core count is only doubling every two years, and DRAM DIMM capacity is doubling only every three years. Consequently, the memory capacity per core expected is to drop by 30% every two years with worse trends for memory bandwidth. The storage bandwidth is also doubling once in two years. As a result, the widening gap between the rate of data generation and the capabilities of computing elements is forcing both academic and industrial researchers to explore alternative memory and storage technologies that can significantly increase the capacity, provide higher data access bandwidth and lower latency. Several prior industrial and academic research have studied promising memory and storage technologies such as byte addressable nonvolatile memory, stacked 3D-DRAM [52, 92, 102, 53]. However, each of these technologies has advantages and disadvantages. For instance, byte addressable NVMs offer 10x higher memory capacity than DRAM but suffer from 5x higher write latency and bandwidth cost. Similarly, on-chip stacked 3D-DRAM [52, 92, 102, 53] increase memory bandwidth by 8x-14x [52], but are expected to have a very limited capacity. To provide applications with large capacity, low latency, and high bandwidth, future systems will support heterogeneous memory. The focus of this dissertation was to understand the implication of heterogeneous memory on the system software, applications, and other system resources such as cache and energy. We next summarize the findings.

9.1.1 Summary

In the first part of this dissertation, we analyzed the impact of heterogeneous memory on applications and discussed the limitations of state-of-the-art solutions, which support heterogeneous memory such as NVM using a virtual file system (VFS) for scaling memory and persistent storage. VFS-based solutions cannot automatically scale across multiple memory types and are cache and TLB inefficient. To address these drawbacks, we designed persistent virtual memory (pVM), a system that treats NVMs as a virtual memory and extends the virtual memory data structures. With a virtual memory abstraction, pVM provides automatic OS-level memory capacity scaling, flexible memory placement policies, and fast persistent object storage. pVM's automatic capacity scaling improved application performance by 2x, reduced TLB and cache miss by up to 80%, and reduced OS-level cost for persistent storage by 4x.

In the second part of this dissertation, we addressed the heterogeneous memory support for virtualized environment. Most large scale applications run in a public or private cloud which is virtualized. When we started this research, we observed that unlike a bare-metal OS, in most virtualized environments, applications do not have even support for NUMA node-specific memory placement policies. Unlike DRAM-based NUMA systems, incorrect data placement in a heterogeneous memory NUMA can have detrimental SLA and monetary impact. Hence, we designed HeteroOS to provide memory heterogeneity-awareness to the guest-OS. We showed that by exposing awareness, heterogeneous memory placement could be done transparently, without the application support, by extracting the information about how applications use memory pages. We also provided a guest-OS and hypervisor-level coordinated management that improved the performance by 2x over the state-of-the-art approaches that managed everything in the hypervisor.

Increasing the number of memory types and devices will increase the processor cache sharing. In the third part of this dissertation, we discussed the system resource usage implications of heterogeneous memory. Memory types such as NVM have higher data write latency and energy, and therefore use the processor cache to avoid direct writes. Dual use

of NVM for both capacity and persistence increases cache sharing conflicts impacting application efficiency. To overcome this, we designed a novel OS-level contiguous memory page allocation method that partitions the cache for persistence and capacity use and reduces misses by 10%. We also optimized the NVM persistent memory allocator to reduce NVM writes resulting in a 12% lower cache misses. We also showed that using strict consistency and durability can increase CPU and NVM energy usage by up to 7.1x and 4.2x, respectively. Hence, we proposed an energy efficient and relaxed durability that reduces the CPU and NVM energy usage by up to 3x and 2x, respectively.

In the fourth part of this dissertation, we explored the additional benefits of redesigning the application-level software for memory heterogeneity. We first studied the checkpoint-restart functionality of HPC applications. Instead of using NVM as a disk, we redesigned checkpoint-restart to use NVM as a virtual memory. Our results showed 15% faster checkpoints and 45% lower interconnect bandwidth usage for remote checkpoints. We next examined the benefits of using NVM as a virtual memory to improve the storage performance of web browsers that run in a sandboxed environment. Our technique of using the memory page protection technique for NVM provides two benefits. (i) Applications can use NVM for a secured memory-based storage that avoids filesystem APIs in a sandboxed environment. As result of avoiding expensive filesystem APIs in a sandboxed environment, the storage performance improved by 2x. We finally redesigned the data structure of a NoSQL database based on the log-structured merge tree (LSM). Specifically, we replaced the existing disk-based persistent storage structures to a persistent memory-based data structures for exploiting byte addressability and also exploit parallelism in data access. As a result, the throughput improved by 3x.

This dissertation provides hard evidence that new OS and hypervisor-level system design principles are required for an efficient and application-transparent heterogeneous memory management. Next, to address the extra system resource usage challenges of memory heterogeneity, this dissertation shows the importance of novel cache- and energy-efficient system software principles. Finally, this dissertation demonstrates the opportunities for an application-level redesign for achieving optimal benefits from memory heterogeneity.

9.1.2 Lessons Learned

We next present a summary of general lessons we learned while working on this dissertation.

9.1.2.1 The need for a unified OS-level abstraction

"All problems in computer science can be solved by another level of indirection," is a famous quote attributed to Butler Lampson. Although abstractions help to simplify the complexity of application interfaces and support legacy code, every the level of software indirection also adds additional latency by moving applications one steps away from direct hardware access.

In the initial stages of our research, we mainly concentrated our research in understanding the implications of byte addressable interface for applications by treating NVM storage as a heap. However, our OS-level support used the virtual filesystem support for NVM [67]. In fact, the benefits of using an object or heap interface for persistent storage have been well documented by the earliest systems – Rio Vista [103], and RVM [136]. Rio Vista and LRVM proposed the heap-based interface for object storage in applications and at the OS layer used a combination of block-based storage and battery-backed RAM. Also, Rio Vista provides persistence guarantees only at log flushes. Recent research such as [146, 56] also embraced persistent heap programming models and optimized the interface specifically for byte addressable NVM with features like processor cache flush on data commit. These new systems also use the VFS-backed memory. OSes such as Linux, Osx, and Windows also provide a high-level memory-mapped storage abstraction to reduce the cost of system calls and exceptions.

While byte addressable heap-like interface at the application layer improves performance, there has been a little focus on the OS-level abstractions, specifically, for storage technologies that resemble closer to memory. Based on the experience gained during this dissertation, we argue that lack of a unified abstraction for byte addressable heterogeneous memory such as NVM at the OS creates several limitations such as the lack of seamless capacity scaling and more importantly, the inability to reuse of several OS-level functionalities designed for memory devices.

Linux POSIX interface provides a unified *mmap()* interface to applications that can

map any device into the application's address space including DRAM, NVM, block storage device, and other hardware peripherals. However, in the OS, the devices and peripherals are managed independently using their customized OS subsystem. Using a separate subsystem is only useful when the devices significantly vary in their hardware characteristics and their purpose of use. For example, DRAM provides byte addressable access for storing volatile data in the form of pages, whereas an SSDs provides block-based access for storing persistent data. Using an entirely different subsystem for devices with completely different hardware characteristics provides a clear separation and avoids unnecessary complexities.

NVMs, in contrast, resembles a DRAM more closely compared to disks. Hence, for using NVMs for scaling memory capacity along with DRAM, it is important to align the management of NVMs with DRAMs using a single subsystem. We believe, this holds true even for other future memory types such as stacked 3D-DRAM. We learned the following from our experience of integrating NVMs with the virtual memory subsystem.

- The virtual memory subsystem of existing OSes such as Linux is structured and generic that makes it easier to integrate different memory types. Specifically, the "memory as NUMA node" abstraction provides flexibility for integrating different memory devices without requiring significant changes to the virtual memory.

- New memory management policies can be easily added and applied to a particular set of memory nodes. In our pVM design, adding NVM-specific policies required less than a week of implementation.

- Virtual memory subsystem has undergone decades of optimization for making them TLB and cache efficient. Using the virtual memory APIs and data structures allowed us to inherit most of these virtues seamlessly. This includes the resource management, garbage collection, and high-performance locking and synchronization benefits.

- Finally, providing an OS-level unified abstraction provided application transparent management avoiding the need for developers to write code for managing different memory types.

9.1.2.2 Impact of reducing the performance gap between memory and storage.

New memory and storage technologies show a great promise in reducing the gap between volatile memory and storage performance. As a result, decades-old assumption about hiding the storage behind memory, and shielding direct access to the storage device via intermediate memory buffers is no longer required. In fact, using memory only adds a layer of software indirection, thereby increasing the software latency for storage access. In this entire dissertation, we learned that directly exposing memory-based NVM storage as a byte addressable memory placed in parallel with DRAM provides following benefits.

Faster random access can avoid serialization cost. Byte addressable heterogeneous memory such as NVM also provide the benefit of fast random access, unlike disk-based system. In disk-based systems, the location of data, and the data access pattern – sequential vs. random – is an important factor for performance. However, in NVMs, random access cost of data is approximately 100x cheaper compared to the disk-based system. As a result, the traditional approach of serializing data – converting in-memory data or data structures to disk-friendly blocks or the vice versa (deserialization) is not required.

Redesign of persistent and nonpersistent data structures. OS and application-level data structures can be broadly classified into (i) in-memory data structures which are strictly designed for volatile heaps, and (ii) persistent data structures that consider the fact that either a part of the data structure or the full data structure is stored in a block-based device. In persistent data structures, updates or access algorithms aim to reduce random access to a block-based persistent device. Current OSes provide extensive optimizations to both forms of data structures. For example, the virtual memory management extensively uses in-memory forms such as RB-tree (Red-black trees), hashtable, or binary search trees, whereas the filesystems extensively use the disk-friendly B-trees. Note that, designing efficient persistent data structure is complex and requires additional processor cycles to manage the data structures between their persistent (disk) and nonpersistent (in-memory) representation. These observations also apply to different application-level data structures.

However, with the introduction of NVMs, that can provide fast, persistent, byte-addressable random access, there is a new opportunity for redesigning OS and application persistent data structures for byte addressable memory-based persistence at a close to hardware speeds. Also, there is a new opportunity for making some of the non-persistent data structures persistent friendly with a simple extension that does not impact the performance or space complexity. In this dissertation, we redesigned log-structured merge trees by extending their in-memory skiplist to support persistence. Our results showed both performance benefits and the ability to reuse data structure optimizations. However, we believe a lot of research for data structure redesign is yet to be done.

9.1.2.3 The importance of memory placement across OS subsystems

When we first started exploring an automatic OS-level memory placement mechanism for heterogeneous memory, we only considered the application's user-level heap memory pages without taking into account the placement of memory pages at the OS-level. The rationale behind this approach was that faster memories have limited capacity, and hence it is important to prioritize the heap which constitutes 90% of applications' memory. Although, placing the heap memory pages to a faster memory provided performance benefits, for storage-intensive applications such as a database and graph computations, and network-intensive applications such as a key-value store, the benefits from memory heterogeneity and OS-level placement were marginal. A closer examination showed a significantly higher memory access of the OS allocated pages which include the buffer cache, the page cache, and the OS network buffers. In fact, these pages had a high page reuse and placing them in a faster memory provided a significant performance boost. In a traditional homogeneous memory OSes, the heap pages are always prioritized, and whenever there is a lack of free memory pages, the OS allocated buffer and page caches are swapped. In contrary, for heterogeneous memory systems, we observed that equally prioritizing the heap and OS-level memory pages based on their access pattern is important for achieving maximum benefits from memory heterogeneity.

Hardware support. As extensively discussed in this dissertation, hardware support for memory placement in a heterogeneous memory system is critical for avoiding software complexity. The hardware support includes exposing useful hardware-level counters and instructions to query about the hardware (memory) usage information by the software. Specifically, as we discussed earlier, information to the software such as hotness of a page, the overall memory bandwidth usage and the impact of incorrect memory placement on the applications can significantly improve the accuracy of memory placement mechanisms. Take the case of NVMs, which suffer 5x higher write latency but have almost the same read latency as DRAM. A page-level information about the total number of reads vs. writes can introduce a new paradigm of how we design a future data structure for NVMs.

9.2 Future work

Memory heterogeneity is bound to reduce the gap between compute, memory, and storage mediums. As a result, several traditional OS, application, and theoretical a ssumptions about how we design computer programs is bound to change. As this thesis describes, an end-to-end vertical approach is required for exploiting the maximum performance, reliability, and energy benefits from memory heterogeneity. The insights gathered through this dissertation work suggest several promising directions.

Impact of heterogeneous memory on data parallel computations. Data parallel computations such as deep learning require 100s to 1000s of machines to analyze and train petabytes of data. However, limited DRAM capacity can severely impact application performance and more importantly, accuracy of these applications. Future memory technologies such as stacked-3D DRAM and NVM with high bandwidth and large capacity respectively, can be very effective in addressing the capacity limitations of DRAM. Further, NVMs with their virtue of persistence can substantially reduce the communication and synchronization overhead with fast, and durable transactions.

OS subsystems and application datastructure redesign. Although this dissertation provides support for future heterogeneous memory, there is an opportunity to revisit and redesign commonly used OS-level and application-level algorithmic data structures that

consider multiple levels of memory. For example, current OS data structures such as red-black trees that are extensively used for maintaining virtual memory pages as discussed in Chapter 3 may no longer scale for substantially large memory. Additionally, apart from the network and storage stack considered in this thesis, several opportunities exist for supporting other subsystems of the OS.

Improving the robustness of future systems. Hardware heterogeneity in current platforms concerning their computer, memory, storage, and communication resources, makes the systems highly configurable to the needs of different applications.However, in large-scale systems, heterogeneity increases the complexity of fault tolerance. Although memory technologies such as NVM can scale, however, the device durability is significantly lower compared to the DRAM. Hence, for the efficient use of heterogeneous memory, it is important to design system software that can improve the robustness of future systems.

Many additional exciting research problems exists at the system level, and at the intersection with other areas. We will contribute the software artifacts to the open source community and enable additional exploration on topics by the broader community.

REFERENCES

[1] http://www.canonware.com/jemalloc.

[2] "Computer language bench." http://tinyurl.com/b29bd2j.

[3] "Facebook RocksDB." https://github.com/facebook/rocksdb.

[4] "Google LevelDB ." http://tinyurl.com/osqd7c8.

[5] "HTML5 Storage." http://bit.ly/o8g5vm.

[6] "Intel Atom - Xolo." http://www.xolo.in/.

[7] "Intel-Micron Memory 3D XPoint." http://intel.ly/1eICR0a.

[8] "Intel RAPL driver." http://lwn.net/Articles/545745/.

[9] "JPEG library." http://libjpeg.sourceforge.net/.

[10] "Lammps benchmark configuration." http://www.sandia.gov/benchmarks/.

[11] "LANL Memcpy Benchmark." http://tinyurl.com/92nvgjd.

[12] "Linux Scalability Benchmark." http://lse.sourceforge.net/.

[13] "MacSim- Simulator for heterogeneous architecture." http://code.google.com/p/macsim/.

[14] "Memcached." http://memcached.org/.

[15] "MongoDB." https://docs.mongodb.com/.

[16] "Nginx memory usage." https://www.nginx.com/blog/nginx-websockets-performance/.

[17] "NGinx Webserver." http://nginx.org.

[18] "Numonyx PCM projection." http://bit.ly/1sph5Qz.

[19] "OpenCV." http://opencv.org/.

[20] "Redis." http://redis.io/.

[21] "S3fuse filesystem." https://github.com/s3fs-fuse/s3fs-fuse.

[22] "Slow browser i/o." http://mzl.la/x55dKq.

[23] "Snappy Compession." http://tinyurl.com/ku899co.

[24] "SQLite." http://www.sqlite.org.

[25] "Ulrich Drepper. "What every programmer should know about memory,"." www.akkadia.org/drepper/cpumemory.pdf.

[26] "VMWare vNUMA." https://www.vmware.com/files/pdf/techpaper/VMware-vSphere-CPU-Sched-Perf.pdf.

[27] "Webshoot Bench." http://bit.ly/18XTihb.

[28] "WebShootOut browser benchmark." bit.ly/19qkSnr.

[29] Abbasi, H., Wolf, M., Eisenhauer, G., Klasky, S., Schwan, K., and Zheng, F., "Datastager: scalable data staging services for petascale applications," in *Proceedings of the 18th ACM international symposium on High performance distributed computing*, HPDC '09, (New York, NY, USA), ACM.

[30] Akel, A., Caulfield, A. M., Mollov, T. I., Gupta, R. K., and Swanson, S., "Onyx: A protoype phase change memory storage array," in *Proceedings of the 3rd USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'11, (Berkeley, CA, USA), pp. 2–2, USENIX Association, 2011.

[31] Akin, B., Franchetti, F., and Hoe, J. C., "Data reorganization in memory using 3d-stacked dram," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, (New York, NY, USA), pp. 131–143, ACM, 2015.

[32] Amur, H., Richter, W., Andersen, D. G., Kaminsky, M., Schwan, K., Balachandran, A., and Zawadzki, E., "Memory-efficient groupby-aggregate using compressed buffer trees," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, (New York, NY, USA), pp. 18:1–18:16, ACM, 2013.

[33] Ananian, C. S. and Rinard, M., "Efficient object-based software transactions," in *SCOOL '05*.

[34] Arulraj, J., Pavlo, A., and Dulloor, S. R., "Let's talk about storage & recovery methods for non-volatile memory database systems," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, (New York, NY, USA), pp. 707–722, ACM, 2015.

[35] Avissar, O., Barua, R., and Stewart, D., "An optimal memory allocation scheme for scratch-pad-based embedded systems," *ACM Trans. Embed. Comput. Syst.*, vol. 1, pp. 6–26, Nov. 2002.

[36] Badam, A. and Pai, V. S., "Ssdalloc: Hybrid ssd/ram memory management made easy," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, (Berkeley, CA, USA), pp. 211–224, USENIX Association, 2011.

[37] Bautista-Gomez, L., Tsuboi, S., Komatitsch, D., Cappello, F., Maruyama, N., and Matsuoka, S., "Fti: high performance fault tolerance interface for hybrid systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, (New York, NY, USA), ACM, 2011.

[38] Bent, J., Gibson, G., Grider, G., McClelland, B., Nowoczynski, P., Nunez, J., Polte, M., and Wingate, M., "Plfs: a checkpoint filesystem for parallel applications," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09.

[39] Bergman, K. and et al., S. B., "Exascale computing study," 2008.

[40] Bhattacharjee, A. and Martonosi, M., "Characterizing the tlb behavior of emerging parallel workloads on chip multiprocessors," in *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, (Washington, DC, USA), pp. 29–40, IEEE Computer Society, 2009.

[41] Bienia, C., *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton, NJ, USA, 2011. AAI3445564.

[42] Blasing, T., Batyuk, L., Schmidt, A.-D., Camtepe, S., and Albayrak, S., "An android application sandbox system for suspicious software detection," in *Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on*, pp. 55–62, 2010.

[43] Borkar, S., "Designing reliable systems from unreliable components: The challenges of transistor variability and degradation," *IEEE Micro*, vol. 25, pp. 10–16, Nov. 2005.

[44] Boyd-Wickizer, S., Chen, H., Chen, R., Mao, Y., Kaashoek, F., Morris, R., Pesterev, A., Stein, L., Wu, M., Dai, Y., Zhang, Y., and Zhang, Z., "Corey: An operating system for many cores," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, (Berkeley, CA, USA), pp. 43–57, USENIX Association, 2008.

[45] Brad Chen, David Sehr, N. F., "Native client: Accelerating web applications." `http://tinyurl.com/mhbz59`.

[46] Cattell, R., "Scalable sql and nosql data stores," *SIGMOD Rec.*, vol. 39, pp. 12–27, May 2011.

[47] Caulfield, A. M., De, A., Coburn, J., Mollow, T. I., Gupta, R. K., and Swanson, S., "Moneta: A high-performance storage array architecture for next-generation, non-volatile memories," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, (Washington, DC, USA), pp. 385–395, IEEE Computer Society, 2010.

[48] Caulfield, A. M., Mollov, T. I., Eisner, L. A., De, A., Coburn, J., and Swanson, S., "Providing safe, user space access to fast, solid state disks," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, (New York, NY, USA), pp. 387–400, ACM, 2012.

[49] "CEPH file-systems." `https://ceph.com/ceph-storage/file-system/`.

[50] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E., "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, pp. 4:1–4:26, June 2008.

[51] Chang, J. and Sohi, G. S., "Cooperative cache partitioning for chip multiprocessors," in *Proceedings of the 21st Annual International Conference on Supercomputing*, ICS '07, (New York, NY, USA), pp. 242–252, ACM, 2007.

[52] Chou, C.-C., Jaleel, A., and Qureshi, M., "Batman: Maximizing bandwidth utilization for hybrid memory systems," in *Technical Report, TR-CARET-2015-01 (March 9, 2015)*, 2015.

[53] Chou, C., Jaleel, A., and Qureshi, M. K., "Cameo: A two-level memory organization with capacity of main memory and flexibility of hardware-managed cache," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, (Washington, DC, USA), pp. 1–12, IEEE Computer Society, 2014.

[54] Coburn, J., Bunker, T., Gupta, R. K., and Swanson, S., "From aries to mars: Reengineering transaction management for next-generation, solid-state drives," in *UCSD CSE Technical Report*, 2012.

[55] Coburn, J., Bunker, T., Schwarz, M., Gupta, R., and Swanson, S., "From aries to mars: Transaction support for next-generation, solid-state drives," in *SOSP 2013*, pp. 197–212.

[56] Coburn, J., Caulfield, A. M., Akel, A., Grupp, L. M., Gupta, R. K., Jhala, R., and Swanson, S., "Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, (New York, NY, USA), pp. 105–118, ACM, 2011.

[57] Condit, J., Nightingale, E. B., Frost, C., Ipek, E., Lee, B., Burger, D., and Coetzee, D., "Better i/o through byte-addressable, persistent memory," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, (New York, NY, USA), pp. 133–146, ACM, 2009.

[58] Constandache, I., Gaonkar, S., Sayler, M., Choudhury, R., and Cox, L., "Enloc: Energy-efficient localization for mobile phones," in *INFOCOM 2009*, pp. 2716–2720, 2009.

[59] Deng, Q., Meisner, D., Ramos, L., Wenisch, T. F., and Bianchini, R., "Memscale: Active low-power modes for main memory," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, (New York, NY, USA), pp. 225–238, ACM, 2011.

[60] Dhiman, G., Ayoub, R., and Rosing, T., "Pdram: a hybrid pram and dram main memory system," DAC '09, (New York, NY, USA).

[61] Dillon, M., "Design elements of the freebsd vm system." `http://tinyurl.com/kftjzqy`.

[62] Dong, X., Muralimanohar, N., Jouppi, N., Kaufmann, R., and Xie, Y., "Leveraging 3d pcram technologies to reduce checkpoint overhead for future exascale systems," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, (New York, NY, USA), ACM, 2009.

[63] Dong, X., Xie, Y., Muralimanohar, N., and Jouppi, N. P., "Hybrid checkpointing using emerging nonvolatile memories for future exascale systems," *ACM Trans. Archit. Code Optim.*

[64] Dong, X., Xie, Y., Muralimanohar, N., and Jouppi, N. P., "Simple but effective heterogeneous main memory with on-chip memory controller support," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, (Washington, DC, USA), pp. 1–11, IEEE Computer Society, 2010.

[65] Duan, R., Bi, M., and Gniady, C., "Exploring memory energy optimizations in smartphones," in *Proceedings of the 2011 International Green Computing Conference and Workshops*, IGCC '11, (Washington, DC, USA), IEEE Computer Society, 2011.

[66] Duan, R., Bi, M., and Gniady, C., "Exploring memory energy optimizations in smartphones," in *Green Computing Conference and Workshops (IGCC), 2011 International*, pp. 1–8, IEEE, 2011.

[67] Dulloor, S. R., Kumar, S., Keshavamurthy, A., Lantz, P., Reddy, D., Sankaran, R., and Jackson, J., "System software for persistent memory," in *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, (New York, NY, USA), pp. 15:1–15:15, ACM, 2014.

[68] Dulloor, S. R., Roy, A., Zhao, Z., Sundaram, N., Satish, N., Sankaran, R., Jackson, J., and Schwan, K., "Data tiering in heterogeneous memory systems," in *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, (New York, NY, USA), pp. 15:1–15:16, ACM, 2016.

[69] Ferdman, M., Adileh, A., Kocberber, O., Volos, S., Alisafaee, M., Jevdjic, D., Kaynak, C., Popescu, A. D., Ailamaki, A., and Falsafi, B., "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," in *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, (New York, NY, USA), pp. 37–48, ACM, 2012.

[70] Gal, A., "Firefox compartments." `http://bit.ly/qOTFid`.

[71] Gallagher, A. C. and Chen, T., "Clothing cosegmentation for recognizing people," in *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, pp. 1–8, June 2008.

[72] Ghodsi, A., Zaharia, M., Hindman, B., Konwinski, A., Shenker, S., and Stoica, I., "Dominant resource fairness: Fair allocation of multiple resource types," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, (Berkeley, CA, USA), pp. 323–336, USENIX Association, 2011.

[73] Giles, E., Doshi, K., and Varman, P., "Bridging the programming gap between persistent and volatile memory using wrap," in *CF '13*.

[74] Google, "LevelDB." `http://bit.ly/1wEklYD`.

[75] Google, "LevelDB benchmark." http://bit.ly/1ERE4u7.

[76] Gorman, M., "Understanding the Linux Virtual Memory Manager." http://bit.ly/1n1xIhg.

[77] Graefe, G., "Write-optimized b-trees," in *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pp. 672–683, VLDB Endowment, 2004.

[78] Guerra, J., Marmol, L., Campello, D., Crespo, C., Rangaswami, R., and Wei, J., "Software persistent memory.," in *USENIX Annual Technical Conference*, pp. 319–331, 2012.

[79] Gupta, V., Lee, M., and Schwan, K., "Heterovisor: Exploiting resource heterogeneity to enhance the elasticity of cloud platforms," in *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '15, (New York, NY, USA), pp. 79–92, ACM, 2015.

[80] Gutierrez, A., Cieslak, M., Giridhar, B., Dreslinski, R. G., Ceze, L., and Mudge, T., "Integrated 3d-stacked server designs for increasing physical density of key-value stores," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, (New York, NY, USA), pp. 485–498, ACM, 2014.

[81] Hähnel, M., Döbel, B., Völp, M., and Härtig, H., "Measuring energy consumption for short code paths using rapl," *SIGMETRICS Perform. Eval. Rev.*, vol. 40, pp. 13–17, Jan. 2012.

[82] Hanson, H. and Rajamani, K., "What computer architects need to know about memory throttling," in *Proceedings of the 2010 International Conference on Computer Architecture*, ISCA'10, (Berlin, Heidelberg), pp. 233–242, Springer-Verlag, 2012.

[83] Harter, T., Dragga, C., Vaughn, M., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H., "A file is not a file: Understanding the i/o behavior of apple desktop applications," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, (New York, NY, USA), pp. 71–83, ACM, 2011.

[84] Heo, J., Zhu, X., Padala, P., and Wang, Z., "Memory overbooking and dynamic control of xen virtual machines in consolidated environments," in *2009 IFIP/IEEE International Symposium on Integrated Network Management*, pp. 630–637, June 2009.

[85] Hertz, M. and Berger, E. D., "Quantifying the performance of garbage collection vs. explicit memory management," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, (New York, NY, USA), pp. 313–326, ACM, 2005.

[86] Intel, "Intel Development Manual." http://intel.ly/1CdHj1r.

[87] Intel, "Logging library." https://github.com/pmem/nvml.

[88] Ipek, E., Condit, J., Nightingale, E. B., Burger, D., and Moscibroda, T., "Dynamically replicated memory: building reliable systems from nanoscale resistive memories," *SIGARCH Comput. Archit. News*, vol. 38, pp. 3–14, Mar. 2010.

[89] Islam, T. Z., Mohror, K., Bagchi, S., Moody, A., de Supinski, B. R., and Eigenmann, R., "Mcrengine: A scalable checkpointing system using data-aware aggregation and compression," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, (Los Alamitos, CA, USA), pp. 17:1–17:11, IEEE Computer Society Press, 2012.

[90] Jaleel, A., "Memory Characterization of Workloads Using Instrumentation-Driven Simulation." `http://bit.ly/15zntbv`.

[91] Jevdjic, D., Volos, S., and Falsafi, B., "Die-stacked dram caches for servers: Hit ratio, latency, or bandwidth? have it all with footprint cache," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, (New York, NY, USA), pp. 404–415, ACM, 2013.

[92] Jiang, X., Madan, N., Zhao, L., Upton, M., Iyer, R., Makineni, S., Newell, D., Solihin, D., and Balasubramonian, R., "Chop: Adaptive filter-based dram caching for cmp server platforms," in *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pp. 1–12, Jan 2010.

[93] Kannan, S., Gavrilovska, A., and Schwan, K., "Reducing the cost of persistence for nonvolatile heaps in end user devices," in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pp. 512–523, Feb 2014.

[94] Kannan, S., Gavrilovska, A., and Schwan, K., "pvm: Persistent virtual memory for efficient capacity scaling and object storage," in *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, (New York, NY, USA), pp. 13:1–13:16, ACM, 2016.

[95] Kannan, S., Gavrilovska, A., Schwan, K., Milojicic, D., and Talwar, V., "Using active nvram for i/o staging," in *Proceedings of the 2nd international workshop on Petascal data analytics: challenges and opportunities*, PDAC '11, (New York, NY, USA), ACM.

[96] Kyrola, A., Blelloch, G., and Guestrin, C., "Graphchi: Large-scale graph computation on just a pc," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, (Berkeley, CA, USA), pp. 31–46, USENIX Association, 2012.

[97] Lai, C., Jiang, S., Yang, L., Lin, S., Sun, G., Hou, Z., Cui, C., and Cong, J., "Atlas: Baidu's key-value storage system for cloud data," in *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–14, May 2015.

[98] Lee, B. C., Ipek, E., Mutlu, O., and Burger, D., "Architecting phase change memory as a scalable dram alternative," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, (New York, NY, USA), pp. 2–13, ACM, 2009.

[99] Li, D., Vetter, J. S., Marin, G., McCurdy, C., Cira, C., Liu, Z., and Yu, W., "Identifying opportunities for byte-addressable non-volatile memory in extreme-scale scientic applications," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, IPDPS '12, ACM, 2012.

[100] Lin, F. X. and Liu, X., "Memif: Towards programming heterogeneous memory asynchronously," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, (New York, NY, USA), pp. 369–383, ACM, 2016.

[101] Liu, R.-S., Shen, D.-Y., Yang, C.-L., Yu, S.-C., and Wang, C.-Y. M., "Nvm duet: Unified working memory and persistent store architecture," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, (New York, NY, USA), pp. 455–470, ACM, 2014.

[102] Loh, G. and Hill, M., "Supporting very large dram caches with compound-access scheduling and missmap," *Micro, IEEE*, vol. 32, pp. 70–78, May 2012.

[103] Lowell, D. E. and Chen, P. M., "Free transactions with rio vista," in *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, (New York, NY, USA), pp. 92–101, ACM, 1997.

[104] Lu, L., Pillai, T. S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H., "Wisckey: Separating keys from values in ssd-conscious storage wisckey: Separating keys from values in ssd-conscious storage," 2016.

[105] Lu, Y., Shu, J., Sun, L., and Mutlu, O., "Loose-ordering consistency for persistent memory," in *Computer Design (ICCD), 2014 32nd IEEE International Conference on*, pp. 216–223, Oct 2014.

[106] Marmol, L., Sundararaman, S., Talagala, N., and Rangaswami, R., "Nvmkv: A scalable, lightweight, ftl-aware key-value store," in *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, (Berkeley, CA, USA), pp. 207–219, USENIX Association, 2015.

[107] Meswani, M., Blagodurov, S., Roberts, D., Slice, J., Ignatowski, M., and Loh, G., "Heterogeneous memory architectures: A hw/sw approach for mixing die-stacked and off-package memories," in *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pp. 126–136, Feb 2015.

[108] Miller, G. A., "Wordnet: a lexical database for english," *Commun. ACM*, vol. 38.

[109] Mogul, J. C., Argollo, E., Shah, M., and Faraboschi, P., "Operating system support for nvm+dram hybrid main memory," HotOS'09, (Berkeley, CA, USA), 2009.

[110] Moody, A., Bronevetsky, G., Mohror, K., and Supinski, B. R. d., "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, (Washington, DC, USA), IEEE Computer Society, 2010.

[111] Moraru, I., Andersen, D. G., Kaminsky, M., Binkert, N., Tolia, N., Munz, R., and Ranganathan, P., "Persistent, protected and cached: Building blocks for main memory data stores," 2012.

[112] Mueller, F., "Compiler support for software-based cache partitioning," in *LCT-RTS 95*.

[113] Narayanan, D. and Hodson, O., "Whole-system persistence," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, (New York, NY, USA), pp. 401–410, ACM, 2012.

[114] Oh, G., Kim, S., Lee, S., and Moon, B., "Sqlite optimization with phase change memory for mobile applications," *PVLDB*, vol. 8, no. 12, pp. 1454–1465, 2015.

[115] O'Neil, P., Cheng, E., Gawlick, D., and O'Neil, E., "The log-structured merge-tree (lsm-tree)," *Acta Inf.*, vol. 33, pp. 351–385, June 1996.

[116] Paul, K. and Kundu, T. K., "Android on mobile devices: An energy perspective," in *Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology*, CIT '10, (Washington, DC, USA), pp. 2421–2426, IEEE Computer Society, 2010.

[117] Pelley, S., Chen, P. M., and Wenisch, T. F., "Memory persistency," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, (Piscataway, NJ, USA), pp. 265–276, IEEE Press, 2014.

[118] Pelley, S., Wenisch, T. F., Gold, B. T., and Bridge, B., "Storage management in the nvram era," *Proc. VLDB Endow.*, vol. 7, pp. 121–132, Oct. 2013.

[119] Phadke, S. and Narayanasamy, S., "Mlp aware heterogeneous memory system," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pp. 1–6, March 2011.

[120] Plank, J. S., Li, K., and Puening, M. A., "Diskless checkpointing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 9, pp. 972–986, Oct. 1998.

[121] Plank, J. S., Luo, J., Schuman, C. D., Xu, L., and Wilcox-O'Hearn, Z., "A performance evaluation and examination of open-source erasure coding libraries for storage," in *Proccedings of the 7th conference on File and storage technologies*, FAST '09, (Berkeley, CA, USA), pp. 253–265, USENIX Association, 2009.

[122] Qian, F., Quah, K. S., Huang, J., Erman, J., Gerber, A., Mao, Z., Sen, S., and Spatscheck, O., "Web caching on smartphones: ideal vs. reality," in *Proceedings of the 10th international conference on Mobile systems, applications, and services*, MobiSys '12, (New York, NY, USA), ACM, 2012.

[123] Qureshi, M. K. and Loh, G. H., "Fundamental latency trade-off in architecting dram caches: Outperforming impractical sram-tags with a simple and practical design," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, (Washington, DC, USA), pp. 235–246, IEEE Computer Society, 2012.

[124] Qureshi, M. K. and Patt, Y. N., "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *MICRO 39*, 2006.

[125] Qureshi, M. K., Srinivasan, V., and Rivers, J. A., "Scalable high performance main memory system using phase-change memory technology," in *Proceedings of the*

36th Annual International Symposium on Computer Architecture, ISCA '09, (New York, NY, USA), pp. 24–33, ACM, 2009.

[126] Radulovic, M., Zivanovic, D., Ruiz, D., de Supinski, B. R., McKee, S. A., Radojković, P., and Ayguadé, E., "Another trip to the wall: How much will stacked dram benefit hpc?," in *Proceedings of the 2015 International Symposium on Memory Systems*, MEMSYS '15, (New York, NY, USA), pp. 31–36, ACM, 2015.

[127] Rajgarhia, A. and Gehani, A., "Performance and extension of user space file systems," in *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, (New York, NY, USA), pp. 206–213, ACM, 2010.

[128] Ramos, L. and Bianchini, R., "Exploiting phase-change memory in cooperative caches," in *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on*, pp. 227–234, Oct 2012.

[129] Ramos, L. E., Gorbatov, E., and Bianchini, R., "Page placement in hybrid memory systems," in *Proceedings of the International Conference on Supercomputing*, ICS '11, (New York, NY, USA), pp. 85–95, ACM, 2011.

[130] Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., and Kozyrakis, C., "Evaluating mapreduce for multi-core and multiprocessor systems," in *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, (Washington, DC, USA), pp. 13–24, IEEE Computer Society, 2007.

[131] Rao, D. S. and Schwan, K., "vnuma-mgr: Managing vm memory on numa platforms," in *2010 International Conference on High Performance Computing*, pp. 1–10, Dec 2010.

[132] Ren, K., Zheng, Q., Patil, S., and Gibson, G., "Indexfs: Scaling file system metadata performance with stateless caching and bulk insertion," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, (Piscataway, NJ, USA), pp. 237–248, IEEE Press, 2014.

[133] Riegel, T., Fetzer, C., and Felber, P., "Time-based transactional memory with scalable time bases," in *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '07, (New York, NY, USA), pp. 221–228, ACM, 2007.

[134] Rientjes, D., "Linux Fake NUMA Patch." `https://www.kernel.org/doc/Documentation/x86/x86_64/fake-numa-for-cpusets`.

[135] Roy, A., Mihailovic, I., and Zwaenepoel, W., "X-stream: Edge-centric graph processing using streaming partitions," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, (New York, NY, USA), pp. 472–488, ACM, 2013.

[136] Satyanarayanan, M., Mashburn, H. H., Kumar, P., Steere, D. C., and Kistler, J. J., "Lightweight recoverable virtual memory," *ACM Trans. Comput. Syst.*, vol. 12, pp. 33–57, Feb. 1994.

[137] Saxena, M. and Swift, M. M., "Flashvm: Virtual memory management on flash," in *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, (Berkeley, CA, USA), pp. 14–14, USENIX Association, 2010.

[138] Schroeder, B. and Gibson, G. A., "Understanding failures in petascale computers," *Journal of Physics: Conference Series*, vol. 78, no. 1, p. 012022, 2007.

[139] Sengupta, D., Wang, Q., Volos, H., Cherkasova, L., Li, J., Magalhaes, G., and Schwan, K., "A framework for emulating non-volatile memory systems with different performance characteristics," in *Proceedings of the Ninth European Conference on Computer Systems*, ICPE '15, ACM, 2015.

[140] Shetty, P., Spillane, R., Malpani, R., Andrews, B., Seyster, J., and Zadok, E., "Building workload-independent storage with vt-trees," in *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, FAST'13, (Berkeley, CA, USA), pp. 17–30, USENIX Association, 2013.

[141] "SNIA persistent memory specs." `http://tiny.cc/o55p4x`.

[142] Soares, L. and Stumm, M., "Flexsc: flexible system call scheduling with exception-less system calls," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, (Berkeley, CA, USA), USENIX Association, 2010.

[143] Suh, G. E., Rudolph, L., and Devadas, S., "Dynamic partitioning of shared cache memory," *J. Supercomput.*, vol. 28, pp. 7–26, Apr. 2004.

[144] Tang, S., Mai, H., and King, S. T., "Trust and protection in the illinois browser operating system," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, (Berkeley, CA, USA), USENIX Association, 2010.

[145] Volos, H., Nalli, S., Panneerselvam, S., Varadarajan, V., Saxena, P., and Swift, M. M., "Aerie: Flexible file-system interfaces to storage-class memory," in *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, (New York, NY, USA), pp. 14:1–14:14, ACM, 2014.

[146] Volos, H., Tack, A. J., and Swift, M. M., "Mnemosyne: Lightweight persistent memory," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, (New York, NY, USA), pp. 91–104, ACM, 2011.

[147] Volos, H., Tack, A. J., and Swift, M. M., "Mnemosyne: lightweight persistent memory," in *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS XVI, (New York, NY, USA), ACM, 2011.

[148] Wahbe, R., Lucco, S., Anderson, T. E., and Graham, S. L., "Efficient software-based fault isolation," in *Proceedings of the fourteenth ACM symposium on Operating systems principles*, SOSP '93, ACM, 1993.

[149] Waldspurger, C. A., "Memory resource management in vmware esx server," *SIGOPS Oper. Syst. Rev.*, vol. 36, pp. 181–194, Dec. 2002.

[150] Waldspurger, C. A., Park, N., Garthwaite, A., and Ahmad, I., "Efficient mrc construction with shards," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, (Berkeley, CA, USA), pp. 95–110, USENIX Association, 2015.

[151] Wang, P., Sun, G., Jiang, S., Ouyang, J., Lin, S., Zhang, C., and Cong, J., "An efficient design and implementation of lsm-tree based key-value store on open-channel ssd," in *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, (New York, NY, USA), pp. 16:1–16:14, ACM, 2014.

[152] Wang, T. and Johnson, R., "Scalable logging through emerging non-volatile memory," *Proc. VLDB Endow.*, vol. 7, pp. 865–876, June 2014.

[153] Wang, Z., Lin, F. X., Zhong, L., and Chishtie, M., "Why are web browsers slow on smartphones?," in *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, HotMobile '11, (New York, NY, USA), ACM, 2011.

[154] Wu, X. and Reddy, A. L. N., "Scmfs: A file system for storage class memory," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, (New York, NY, USA), pp. 39:1–39:11, ACM, 2011.

[155] Wu, X., Xu, Y., Shao, Z., and Jiang, S., "Lsm-trie: An lsm-tree-based ultra-large key-value store for small data," in *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, (Berkeley, CA, USA), pp. 71–82, USENIX Association, 2015.

[156] Yee, B., Sehr, D., Dardyk, G., Chen, J. B., Muth, R., Ormandy, T., Okasaka, S., Narula, N., and Fullagar, N., "Native client: A sandbox for portable, untrusted x86 native code," in *2009 30th IEEE Symposium on Security and Privacy*, pp. 79–93, May 2009.

[157] Yoo, R. M., Romano, A., and Kozyrakis, C., "Phoenix rebirth: Scalable mapreduce on a large-scale shared-memory system," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, (Washington, DC, USA), pp. 198–207, IEEE Computer Society, 2009.

[158] Zhang, X., Dwarkadas, S., and Shen, K., "Hardware execution throttling for multi-core resource management," in *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USENIX'09, (Berkeley, CA, USA), pp. 23–23, USENIX Association, 2009.

[159] Zhang, X., Dwarkadas, S., and Shen, K., "Towards practical page coloring-based multicore cache management," in *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, (New York, NY, USA), pp. 89–102, ACM, 2009.

[160] Zhao, J., Li, S., Yoon, D. H., Xie, Y., and Jouppi, N. P., "Kiln: Closing the performance gap between systems with and without persistence support," in *MICRO '46*, 2013.

[161] Zhao, J., Li, S., Yoon, D. H., Xie, Y., and Jouppi, N. P., "Kiln: Closing the performance gap between systems with and without persistence support," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, (New York, NY, USA), pp. 421–432, ACM, 2013.

[162] Zhao, J., Mutlu, O., and Xie, Y., "Firm: Fair and high-performance memory control for persistent memory systems," in *MICRO-47, 2014*, pp. 153–165.

[163] Zhao, L., Iyer, R., Illikkal, R., and Newell, D., "Exploring dram cache architectures for cmp server platforms," in *Computer Design, 2007. ICCD 2007. 25th International Conference on*, pp. 55–62, Oct 2007.

[164] Zheng, F., Abbasi, H., Docan, C., Lofstead, J., Liu, Q., Klasky, S., Parashar, M., Podhorszki, N., Schwan, K., and Wolf, M., "Predata preparatory data analytics on peta-scale machines," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp. 1–12, April 2010.

[165] Zheng, G., Shi, L., and Kale, L. V., "Ftc-charm++: an in-memory checkpoint-based fault tolerant runtime for charm++ and mpi," in *Proceedings of the 2004 IEEE International Conference on Cluster Computing*, CLUSTER '04, (Washington, DC, USA), IEEE Computer Society.

VITA

Sudarsun Kannan is a PhD candidate at the College of Computing, Georgia Institute of Technology. He is also a member of the NSF-sponsored Center for Experimental Research in Computer Systems (CERCS) which conducts research in the domains of Enterprise, High Performance, and Embedded Systems. His research interests include emerging persistent memories, file systems, virtualization, and large-scale datacenter systems. He holds a Master of Science in Computer Science from Georgia Institute of Technology and a Bachelor of Engineering in Computer Science from Anna University, Chennai, India.